1. Remove Duplicates from a Sorted Array
2. Show Duplicates in a sorted integer Array
3. Delete an element from an integer array
4. Print 1 to N recursively
5. Factorial of a number using recursion
6. Swap Two Numbers
7. Fibonacci Series
8. Find Missing and Duplicate in a sorted integer array
9. Find missing number from 1 to N
10. Reverse an Integer Array
11. Reverse a String
12. Shuffle an integer array
13. Second Largest in an Array
14. Array Rotation
15. Write an algorithm so that a[i]+a[j] = k in a sorted array of integers
16. How to find a first non-repeating character in a String
17. Find out whether two strings are the same or not without using equal() method.
18. String to number conversion
19. String compression
20. Permutation of Integer Array
21. Alternative way to Integer Permutation
22. String permutation
23. Addition of two integers without using arithmetic operators
24. Number Conversion from positive to negative and vice versa
25. Arrays.copyof()
26. Arrays.copyOfRange()

Refined Basic Algorithms

# **Remove Duplicates from a Sorted Array**

**import** java.util.Arrays;

**public** **class** RemoveDuplicatesFromSortedArray {

**public** **static** **int**[] removeDuplicates(**int**[] a) {

**int** j = 0;

**int**[] newArr = **null**;

**for** (**int** i = 0; i < a.length; i++) {

**try** {

**if** ((a[i] ^ a[i + 1]) == 0) {

// Duplicates found

} **else** {

j++;

a[j] = a[i + 1];

}

} **catch** (IndexOutOfBoundsException ibe) {

**break**;

}

}

**int** newSize = (j + 1);

newArr = **new** **int**[newSize];

newArr = Arrays.*copyOf*(a, newSize);

**return** newArr;

}

**public** **static** **int**[] removeDuplicates1(**int**[] a) {

**int** j = 0;

**int**[] newArr = **null**;

**for** (**int** i = 0; i < a.length - 1; i++) {

**if** ((a[i] ^ a[i + 1]) == 0) {

// Duplicates found

} **else** {

j++;

a[j] = a[i + 1];

}

}

**int** newSize = (j + 1);

newArr = **new** **int**[newSize];

newArr = Arrays.*copyOf*(a, newSize);

**return** newArr;

}

**public** **static** **void** main(String[] args) {

**int**[] a = { 6, 7, 8, 9, 9, 10, 11, 11, 11, 12, 13, 13, 14, 14, 14,

14, 15, 15, 17, 19 };

// int[] a = { -3, -2, -2, -2, 0, 0, 1, 1, 2, 3, 4, 5, 5, 5, 6, 6, 6, 6 };

**for** (**int** i : a)

System.***out***.print(i + "\t");

System.***out***.println("\n");

**int**[] c = *removeDuplicates*(a);

System.***out***.println("\n\n");

a = c;

**for** (**int** i : a)

System.***out***.print(i + "\t");

}

}

# **Show Duplicates in a sorted integer Array**

public class DupllicatesInArray {

public static void showDuplicates( int[] a ) {

for( int i=0;i<a.length;i++ ) {

try {

if( **(a[i]^a[i+1]) == 0** )

System.out.println(a[i]);

}

catch (IndexOutOfBoundsException e) { break; }

}

}

public static void main(String[] args) {

int a[]={1,1,1,2,3,4,4,5,6,7,7,9,10};

showDuplicates(a);

}

}

# **Delete an element from an integer array**

**import** java.util.Arrays;

**public** **class** DeleteAnElementFromArray {

**public** **static** **int**[] deleteFromArray(**int**[] a, **int** val) {

**int** j = 0;

**for** (**int** i = 0; i < a.length; i++) {

**try** {

**if** (a[i] == val) {

// Found element

} **else** {

a[j] = a[i];

j++;

}

} **catch** (IndexOutOfBoundsException ibe) {

**break**;

}

}

**int** newSize = j;

**int**[] newArr = Arrays.*copyOf*(a, newSize);

**return** newArr;

}

**public** **static** **void** main(String[] args) {

**int**[] a = { 1, 2, 3, 4, 5, 5, 6, 7,7 };

// It will delete all occurrences of that number

a = *deleteFromArray*(a, 1);

**for** (**int** i : a)

System.***out***.print(i + "\t");

}

}

# **Print 1 to N recursively**

**public** **class** Print1ToN {

**public** **static** **void** print1ToNDescendingOrder(**int** n) {

**if** (n != 0) {

System.***out***.print(n + "\t");

*print1ToNDescendingOrder*(n - 1);

}

}

**public** **static** **void** print1ToNAscendingOrder(**int** n) {

**if** (n != 0) {

*print1ToNAscendingOrder*(n - 1);

System.***out***.print(n + "\t");

}

}

**public** **static** **void** print1ToN(**int** n) {

**if** (n == 0) **return**;

*print1ToN*(n - 1);

System.***out***.print(n + "\t");

}

**public** **static** **void** main(String[] args) {

// print1ToNDescendingOrder(5);//Prints 5 4 3 2 1

*print1ToNAscendingOrder*(5);//prints 1 2 3 4 5

}

}

**Factorial of a number using recursion**

**public class** FactorialOfANumber {  
 **public static int** factorial( **int** n ) {  
  
 **if( n==0 ) return 1;  
 return n\**factorial*( n-1 );**  
 }  
  
 **public static void** main(String[] args) {  
  
 System.***out***.println(*factorial*(6));  
 }  
}

# **Swap Two Numbers**

**int a = 5;  
int b = 6;**

**a = a ^ b;  
b = a ^ b;  
a = a ^ b;**

# **Fibonacci Series**

**public class** FibbonacciSeries {  
 **public static int** fibbo( **int** n ) {  
  
 **if( n == 0 ) return 0;  
 else if( n == 1) return 1;  
 else  
 return *fibbo*( n - 1)+*fibbo*( n - 2 );** }  
 **public static void** main(String[] args) {  
 **int** n = 7;  
 **for** (**int** i = 0; i < n; i++) {  
 System.***out***.println(*fibbo*(i));  
 }  
 }  
}

**Find Missing and Duplicate in a sorted integer array**  
**public class** FindMissingAndDuplicate {  
 **public static void** findMissingAndDuplicate(**int**[] a) {  
 **int n = a.length;  
 int idealSum = ( ( n\*( n+1 ) )/2 );  
 int actualSum = 0;  
 int duplicate = 0;  
 for (int i = 0; i < n; i++) {  
 actualSum = actualSum+a[i];  
 try {  
 if( (a[i] ^ a[i+1]) == 0 )  
 duplicate = a[i];  
 }  
 catch (IndexOutOfBoundsException ibe) {  
 break;  
 }  
 }  
 int missing = idealSum - actualSum+duplicate;** System.***out***.println(**"Duplicate :::"**+duplicate+**"----"**+**"Missing :::"**+missing);  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {1,2,3,5,5,6,7,8,9,10};  
 *findMissingAndDuplicate*(a);  
 }  
}

# **Find missing number from 1 to N**

**public class** FindMissingNo1ToN {  
  
 **public static int** findMissing(**int**[] a, **int** n ) {  
 **int idealSum = ( ( n\*( n+1 ) )/2 );  
 int actualSum = 0;  
 for (int i = 0; i < a.length; i++) {  
 actualSum = actualSum + a[i];  
 }  
 return idealSum - actualSum;** }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {5,4,1,3,2,7};  
 System.***out***.println(**"Missing :::"**+*findMissing*(a,7));  
 }  
}

**Given a list of Integers,Find the tuple of 3 adjacent elements in this list that adds up to the maximum sum.**

Ans : Sort the list in descending order and then pick first m characters.

**Reverse a character array (**Best way to do it**)**

**public static void** reverse1(**char**[] ch) {

**for** (**int** i = 0 , j = ch.**length** - 1 ; i < j ; i++ , j-- ) {  
 **char** temp = ch[i];  
 ch[i] = ch[j];  
 ch[j] = temp;  
 }  
}

**Alternative way**

**public static void** reverse(**char**[] ch) {

**int** mid = ch.**length**/2;  
 **int** len = ch.**length** - 1;

**for (int i = 0; i < mid; i++)** {  
 **char** temp = ch[i];  
 ch[i] = ch[len - i];  
 ch[len - i] = temp;  
 }  
}

# **Reverse an Integer Array**

**public static void** reverseIntArray(**int**[] a) {  
  
 **for (int i = 0 , j = a.length - 1; i < j; i++, j--) {  
 int temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }**}

# **Reverse a String**

**public static** String reverse(String s ) {

**if**( s == **null** || s.length() == 0 ) **return** s;

**return *reverse*(s.substring(1))** + **s.charAt(0)** ;   
}

**Alternative way**

Convert string to character array and reverse the character and form a new String

**Shuffle an integer array**

**import** java.util.Random;

**public** **class** MyShuffle {

**public** **static** **void** main(String[] args) {

**int** a[]={1,2,3,4,5,6,7};

Random random = **new** Random();

**for** (**int** i=0; i<a.length; i++) {

**int** randomPosition = random.nextInt(a.length);

**int** temp = a[i];

a[i] = a[randomPosition];

a[randomPosition] = temp;

}

**for**( **int** k : a )

System.*out*.print(k+" ");

}

}

**Second Largest in an Array**

public class SecondLargestNumberInArray {

public static void secondLargeNumber(int[] arr) {

**int largest = arr[0];**

**int secondLargest = arr[0];**

**for (int i = 0; i < arr.length; i++) {**

**if (arr[i] > largest) {**

**secondLargest = largest;**

**largest = arr[i];**

**} else if (arr[i] > secondLargest) {**

**secondLargest = arr[i];**

**}**

**}**

System.out.println("second largest in array is:" + secondLargest);

}

public static void main(String[] args) {

int arr[] = { 1, 23, 47, 81, 92, 88, 52, 48, 56, 66, 65, 76, 71, 85,

49, 53, 56, 61, 65, 84 };

secondLargeNumber(arr);

}

}

**\*\* In case Nth largest or smallest, sort the array using Quick sort and find the nth position.**

**Array Rotation**

**Rotate Right**

Before : 1 2 3 4 5

After : 5 1 2 3 4

**Rotate Left**

Before : 1 2 3 4 5

After : 2 3 4 5 1

//Rotate Left

**public static void** rotateLeft( Object[] a ) {  
  
 Object firstElement = a[0];  
 **for** (**int** i = 1; i < a.**length**; i++) {  
 a[i-1] = a[i];  
 }  
 a[ a.**length** -1 ] = firstElement;  
}

//Rotate Right  
**public static void** rotateRight( Object[] a ) {  
  
 Object lastElement = a[ a.**length** - 1 ];  
 **for** (**int** i = a.**length** - 1; i > 0 ; i--) {  
 a[i] = a[i - 1];  
 }  
 a[ 0 ] = lastElement;  
}

[Rotating an array by position n](http://codereview.stackexchange.com/questions/69299/rotating-an-array-by-position-n)

Input is 1,2,3,4,5,6,7,8

If rotateOrder = 2

Output should be 7,8,1,2,3,4,5,6

If rotateOrder = 3

Output should be 6,7,8,1,2,3,4,5

Rotate a one-dimensional array of n elements to the right by k steps.   
For instance, with n=7 and k=3, the array {a, b, c, d, e, f, g} is rotated to {e, f, g, a, b, c, d}.

<http://articles.leetcode.com/2010/09/determine-if-binary-tree-is-binary.html>

# Determine if a Binary Tree is a Binary Search Tree (BST)

Write a function isBST(BinaryTree \*node) to verify if a given binary tree is a Binary Search Tree (BST) or not.

First, you must understand the difference between *Binary Tree* and *Binary Search Tree (BST)*. Binary tree is a tree data structure in which each node has at most two child nodes. A binary search tree (BST) is based on binary tree, but with the following additional properties:

* The left subtree of a node contains only nodes with keys less than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* Both the left and right subtrees must also be binary search trees.

This question is a very good interview question, because it *really* tests your understanding of the definition of BST. Most people will fall to this first trap, and gives the following algorithm:

<http://puzzlersworld.com/interview-questions/rotate-and-array-by-k-elements/>

Given an array of n elements, write an algorithm to rotate it right by k element without using any other array. In other words rotate an array in place.  
you can have temp variable to swap two elements. Example is given below.  
**Initial Array: A = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}  
Rotating array A right by 3 elements should result in  
A = {8, 9, 10, 1, 2, 3, 4, 5, 6, 7}  
Hint:** There is a trick, think before doing much of the programming

**So the trick is** to reverse the whole array first then reverse the array from 0 to k-1 and k to n-1.  
For example  
**Array A after reversing whole array: {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}  
Array A after reversing 0 to 2 elements: {8, 9, 10, 7, 6, 5, 4, 3, 2, 1}  
Array A after reversing 3 to 9 elements: {8, 9, 10, 1, 2, 3, 4, 5, 6, 7}**

Complete code is given below.  
public class Rotation {  
  
 private static void swap(int[] a, int i, int j) {  
 int temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 private static void reverse(int[] a, int start, int end) {  
 while (start < end) {  
 *swap*(a, start, end);  
 start++;  
 end--;  
 }  
 }

//Another way to reverse the array

**private static void reverse(int[] a, int start, int end) {  
 for( int i = start , j = end ; i < j ; i++,j-- ) {  
 *int temp = a[i];*  
 a[i] = a[j];  
 a[j] = temp;  
 }  
 }**

**//~~ Rotate Left  
 private static void rotateRight(int[] a, int d) {  
 int n = a.length;  
 *reverse*(a, 0, n - 1);//reverse whole array  
 *reverse*(a, 0, d - 1);//assuming 0 < k < length  
 *reverse*(a, d, n - 1);  
 }**  
 **//~~ Rotate Left  
 /\* Function to left rotate arr[] of size n by d \*/  
 static void rotateLeft(int[] a, int d) {  
 int n = a.length;  
 *reverse*(a, 0, d - 1);  
 *reverse*(a, d, n - 1);  
 *reverse*(a, 0, n - 1);  
 }**

public static void main(String[] args) {  
  
 int[] a = {1, 2, 3, 4, 5, 6, 7, 8, 9};  
 //Rotate right by 3 elements  
 *rotateRight*(a, 3);//7 8 9 1 2 3 4 5 6  
  
 //Rotate left by 3 elements  
 //rotateLeft(a,3);//4 5 6 7 8 9 1 2 3  
  
 for (int i : a)  
 System.*out*.print(i + "\t");  
 }  
}

# **Write an algorithm so that a[i]+a[j] = k in a sorted array of integers**

import java.util.Hashtable;

public class SumOfTwoElementsInArray

{

//Below is the best

public static void printPairs(int[] array, int sum){

Hashtable numberSet= new Hashtable();

for (int i = 0; i < array.length; i++) {

if(numberSet.containsKey(array[i])){

System.out.println("Here is the values "+ array[i] +" and " +numberSet.get(array[i]));

}

else

numberSet.put(sum-array[i], array[i]);

}

}

public static void main(String[] args) {

int[] arr = new int[]{1,2,3,4,5,6,7,9};

printPairs(arr, 7);//This is the best

}

}

**How to find a first non repeating character in a String**

/\*

\* How to find a first non repeating character in a String

\*/

public class FirstNonRepeatedChar {

public static void getfirstNonRepetedChar(char[] ch) {

for (int i = 0; i < ch.length; i++)

{

int flag = 0;

for (int j = 0; j < ch.length; j++)

{

if (ch[i] == ch[j])

{

flag++;

}

if (flag > 2)

{

break;

}

}

if (flag == 1)

{

System.out.print(ch[i] + " ");

}

}

}

public static void main(String[] args) {

char[] ch = { 'A', 'B', 'B', 'C', 'K', 'D', 'A', 'D' };

getfirstNonRepetedChar(ch);

}

}

Output

------

C K

**Alternative**

**public static char** getFirstNonRepeatedChar(String str) {  
 Map<Character, Integer> counts = **new** LinkedHashMap<>(str.length());  
 **for** (**char** c : str.toCharArray()) {  
 counts.put(c, counts.containsKey(c) ? counts.get(c) + 1 : 1);  
 }  
 **for** (Entry<Character, Integer> entry : counts.entrySet()) {  
 **if** (entry.getValue() == 1) {  
 **return** entry.getKey();  
 }  
 }  
 **throw new** RuntimeException(**"didn't find any non repeated Character"**);  
}

**Find out whether two strings are the same or not without using the equality operator not the equals method.**

public class StringEquals {

//Two strings of equal length

public static int isTwoStringsEqual(String s1,String s2) {

int count = 0;

for( int i = 0 ; i < s1.length() ;i++ ) {

**count = count+s1.charAt(i) ^ s2.charAt(i);**

}

return count;

}

public static void main(String[] args) {

String s1 = "abcd";

String s2 = "aqcd";

boolean flag = isTwoStringsEqual(s1, s2) == 0 ? true : false;

System.out.println(flag);

}

}

# **String to number conversion**

public class String2Number {

public static void main(String[] args) {

String s = "1234";

long number=0;

char[] ch = s.toCharArray();

for( int i = 0 ; i < s.length() ; i++ ) {

**number= number\*10 + ( ch[i]-'0' );**

}

System.out.println(number);

}

}

**String compression**

**Input String : aaabbbdccaabcc**

**Output String : a5b4c4d1**

import java.util.Map;

import java.util.TreeMap;

public class StringCompression {

public static void main(String[] args) {

String s = "aaabbbdccaabcc";

char[] ch = s.toCharArray();

Map<Character,Integer> map = new TreeMap<Character,Integer>();

for( int i = 0 ; i < s.length() ; i++ ) {

char token = ch[i];

if( map.containsKey(token )) {

int val = map.get(token);

map.put(token, ++val);

}

else

map.put(token, 1);

}

System.out.println(map);

}

}

**Permutation of Integer Array (***Best way to do it***)**

*//* [*http://introcs.cs.princeton.edu/java/23recursion/Permutations.java.html*](http://introcs.cs.princeton.edu/java/23recursion/Permutations.java.html)**public class** IntegerArrayPermutation {

**public static void** print(**int**[] arr) {  
 **for**( **int** p : arr)  
 System.***out***.print(p + **" "**);  
 System.***out***.println(**"\n"**);  
 }  
  
 **private static void** swap(**int**[] a, **int** i, **int** j) {  
 **int c = a[i];  
 a[i] = a[j];  
 a[j] = c;** }  
  
 **public static void** perm(**int**[] arr) {  
 **int N = arr.length;  
 *perm*(arr, N);** }  
  
 **private static void** perm(**int**[] a, **int** n) {  
 **if (n == 1) {  
 *print*(a);  
 return;  
 }  
 for (int i = 0; i < n; i++) {  
 *swap*(a, i, n-1);  
 *perm*(a, n-1);  
 *swap*(a, i, n-1);  
 }** }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {1,2,3,4};  
 *perm*(a);  
  
 }  
}

**Alternative way to Integer Permutation**

public static void print( int[] arr ) {

for( int i = 0 ; i < arr.length ; i++ )

System.out.print(arr[i]+" ");

System.out.print("\n");

}

private static void swap(int[] a , int i, int j) {

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

public static void **permute**(int[] array, int i) {

if (array.length == i) {

print(array);

return;

}

for (int j = i; j < array.length; j++) {

swap(array, i, j);

permute(array,i+1);

swap(array, i, j);

}

return;

}

**String permutation**

**public** **class** StringPermutation {

**public** **static** **void** perm(String prefix, String s ) {

**int** n = s.length() ;

**if**( n == 0 ) System.*out*.println(prefix);

**else** {

**for**( **int** i = 0 ; i < n ; i++ )

*perm*( **prefix+s.charAt(i) , s.substring(0,i)+s.substring(i+1,n)** );

}

}

**public** **static** **void** main(String[] args) {

*perm*("","abc");

}

}

**Addition of two integers without using arithmetic operators (+, ++, –, -, .. etc)**

public class AddOfTwoNos

{

public static int add(int x, int y)

{

if (y == 0) return x;

else

return add( **x ^ y, (x & y) << 1** );

}

public static void main(String[] args) {

System.out.println(add(10,20));

}

}

**Number Conversion from positive to negative and vice versa**

int x = 27;

**int negativeX = ~x+1;**

System.out.println("Number In Negative ::: "+negativeX);// -27

**int originalX = ~negativeX+1;**

System.out.println("Original or Negative to Positive : "+originalX); //27

**Given a list of Integers,Find the tuple of 3 adjacent elements in this list that adds up to the maximum sum.**

Ans : **Sort the list in descending order and then pick first m characters.**

**Write a thread-safe array-based queue implementation in java**

package com.ddlab.core.algorithm;

import java.util.ArrayList;

/\*

\* Write a thread-safe array-based queue implementation in java.

\* If one thread reaches to a limit, it should wait for dequeue thread to create a space and vice-versa.

\*/

public class ThreadSafeArrayQueue<E> {

private static final int size=10;

private ArrayList<E> arr= new ArrayList<E>(10);

public synchronized void enqueue(E item) throws InterruptedException

{

while(arr.size() == size){

wait();

}

if(arr.isEmpty())

{

notify();

}

System.out.println("Item added : "+item);

arr.add(item);

}

public synchronized E dequeue(int item) throws InterruptedException

{

while(arr.isEmpty())

{

wait();

}

if(arr.size()==size)

{

notify();

}

return arr.remove(0);

}

public static void main(String[] args) throws Exception {

ThreadSafeArrayQueue<String> thSafeQ = new ThreadSafeArrayQueue<String>();

for( int i = 0 ; i < 11 ; i++ )

thSafeQ.enqueue("Item-"+i);

}

}

**Calculate the size of a serialized object**

import java.io.ByteArrayOutputStream;

import java.io.ObjectOutputStream;

import java.util.ArrayList;

import java.util.List;

/\*

\* How to calculate the size of a serialized object

\*/

public class SizeOfAnObject {

public static void main(String[] args) {

try {

List<String> list = new ArrayList<String>();

list.add("ABCD");

ByteArrayOutputStream baos = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(baos);

oos.writeObject(list);

oos.close();

System.out.println("Size of List : " + baos.size());// 65

} catch (Exception e) {

e.printStackTrace();

}

}

}

**SORT R G B in a particular order**

/\*

\* Given a character array as input.

\* Array contains only three types of characters 'R', 'G' and 'B'.

\* Sort the array such that all 'R's comes before 'G's and all 'G's comes before 'B's.

\* Constraint :- No extra space allowed(except O(1) space like variables) and minimise the time complexity.

\*

\* Input: R G B G B G B R R B G R R B B G R

\*

\* Output: R R R R R R G G G G G B B B B B B

\*/

**public** **class** SortRGB {

**public** **static** **void** sortAnArrayInParticularOrder(**char** [] array)

{

**int** indexOfR = 0;

**int** indexOfB = array.length-1;

**for**(**int** i=0;i<=indexOfB;)

{

**if**(array[i]=='R')

{

*swap*(i,indexOfR,array);

indexOfR ++;

i++;

}

**else** **if** (array[i]=='B')

{

*swap*(i,indexOfB,array);

indexOfB --;

}

**else**

{

i++;

}

}

**for**(**char** c:array)

{

System.*out*.print(c+" ");

}

}

**private** **static** **void** swap(**int** i, **int** j, **char**[] array) {

**char** temp = array[i];

array[i]=array[j];

array[j]=temp;

}

**public** **static** **void** main(String[] args) {

String s = "RGBGBGBRRBGRRBBGR";

*sortAnArrayInParticularOrder*(s.toCharArray());

}

}

**System.arraycopy and Arrays.copyOf**

**System.arraycopy**

**The method signature is given below.**

**public static native void** arraycopy(Object src,

**int** srcPos,  
 Object dest,

**int** destPos,  
 **int** length);

**Parameters**

1. **src** -- This is the source array.
2. **srcPos** -- This is the starting position in the source array.
3. **dest** -- This is the destination array.
4. **destPos** -- This is the starting position in the destination data.
5. **length** -- This is the number of array elements to be copied.

Points to be noted

1. Destination array should not be null, ie **int[] b = null**;
2. It is a native method.

Example

**int**[] a = {1,2,3,4,5,6,7,8,9,10};  
**int**[] b = **new int**[a.**length**];

*System.arraycopy(a,0,b,0,a.length);//copy all elements, 1 2 3 4 5 6 7 8 9 10*

System.*arraycopy*(a,0,b,0,2);*//copy 2 elements , 1 2 0 0 0 0 0 0 0 0*

**Arrays.copyOf**

**The method structure is given below.**

**public static int**[] copyOf(**int**[] original, **int** newLength)

**Parameters**

**Original – pass the actual array.**

**newLength – pass the number of elements to copy**

The source code is given below.

**public static int**[] copyOf(**int**[] original, **int** newLength) {  
 **int**[] copy = **new int**[newLength];  
 System.*arraycopy*(original, 0, copy, 0,  
 Math.*min*(original.**length**, newLength));  
 **return** copy;  
}

Arrays.copyOf always creates a new array and internally uses System.arrayCopy.

An example is given below.

**int**[] a = {1,2,3,4,5,6,7,8,9,10};

**int**[] b = **null**;

b = Arrays.*copyOf*(a, a.**length**); //1 2 3 4 5 6 7 8 9 10

b = Arrays.*copyOf*(a, 3); // 1 2 3 , here the size of the array is 3, not the 10.

**Arrays.copyOfRange()**

**public static int[] copyOfRange(int[] original, int from, int to)**

The source code is given below.

**public static int**[] copyOfRange(**int**[] original, **int** from, **int** to) {  
 **int** newLength = to - from;  
 **if** (newLength < 0)  
 **throw new** IllegalArgumentException(from + **" > "** + to);  
 **int**[] copy = **new int**[newLength];  
 System.*arraycopy*(original, from, copy, 0,  
 Math.*min*(original.**length** - from, newLength));  
 **return** copy;  
}

parameters

**original – Actual array**

**from – starting position , initial index of the array, including**

**to – ending position, final index of the array, excluding**

**Example is given below.**

**int**[] a = {1,2,3,4,5,6,7,8,9,10};

**int**[] b = **null**;

b = Arrays.*copyOfRange*(a, 2,5); // 3 4 5

So here

Starting position - a[2] ie 3

Ending position – a[5] ie 6, but it will copy upto a[4}, as mentioned in the document as excluding.

Sorting and Searching Data Structure

**Quick Sort in Ascending Order**

**Time Complexity: Best Case and Average Case : O(*n* log *n*) , Worst Case : O(*n*2)**

**Space Complexity: Worst Case : O(log *n*)**

**public class** QuickSortAscending {*//Sorting in Ascending order*  
 **public static void** quickSort(**int**[] a) {  
 ***quickSort*(a, 0 , a.length - 1 );** }  
  
 **public static void** quickSort(**int**[] a , **int** low , **int** high ) {  
  
 **int** i = low;  
 **int** j = high;  
 **int** pivot = a[low+ (high - low )/2 ];  
  
 **while( i <= j ) {  
  
 while( a[i] < pivot ) i++;  
  
 while( a[j] > pivot ) j--;  
  
 if( i <= j ) {  
 *swap*(a , i , j);  
 i++;  
 j--;  
 }  
 }  
 if( i < high ) {  
 *quickSort*( a, i , high );  
 }  
 if( j > low )  
 *quickSort*( a,low,j );**  
 }  
  
 **public static void** swap(**int**[] a, **int** i , **int** j ) {  
 **int** temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {24, 3, 45, 20, 56, 75, 2, 56, 99, 53, 12};  
 *quickSort*(a);  
 **for**( **int** i : a )  
 System.***out***.print(i + **"\t"**);  
 }  
} output -> 2 3 12 20 24 45 53 56 56 75 99

For Descending Order

**while( a[i] > pivot ) i++;**

**while( a[j] < pivot ) j--;**

**Quick Sort in Descending Order**

*//Descending Order***public class** QuickSortDescending {  
  
 **public static void** quickSort(**int**[] a) {  
  
 *quickSort*(a, 0, a.**length** - 1);  
 }  
  
 **public static void** quickSort(**int**[] a, **int** low, **int** high) {  
  
 **int** i = low;  
 **int** j = high;  
 **int** pivot = a[low + (high - low) / 2];  
  
 **while (i <= j) {  
  
 while (a[i] > pivot) i++;  
  
 while (a[j] < pivot) j--;  
  
 if (i <= j) {  
 *swap*(a, i, j);  
 i++;  
 j--;  
 }  
 }  
 if (i < high) {  
 *quickSort*(a, i, high);  
 }  
 if (j > low)  
 *quickSort*(a, low, j);** }  
  
 **public static void** swap(**int**[] a, **int** i, **int** j) {  
 **int** temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {24, 3, 45, 20, 56, 75, 2, 56, 99, 53, 12};  
 *quickSort*(a);  
 **for** (**int** i : a)  
 System.***out***.print(i + **"\t"**);  
 }  
}

output -> 99 75 56 56 53 45 24 20 12 3 2

**Trick for Ascending and Descending order in Quick Sort**

**Descending Order**

**while (a[i] > pivot) i++;  
  
while (a[j] < pivot) j--;**

**Ascending Order**

**while( a[i] < pivot ) i++;  
  
while( a[j] > pivot ) j--;**

To find the Nth Greatest or Smallest, sort the array using Quick sort and find the index of the array as shown below.

**public static int getNthSmallest(int[] a , int n) {  
 *quickSort*(a);  
 return a[n];  
}**

**Order and Space Complexity Table**

**Space Complexity**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Timsort** | [**Introsort**](https://en.wikipedia.org/wiki/Introsort) | [**Merge sort**](https://en.wikipedia.org/wiki/Merge_sort) | [**Quicksort**](https://en.wikipedia.org/wiki/Quicksort) | [**Insertion sort**](https://en.wikipedia.org/wiki/Insertion_sort) | [**Selection sort**](https://en.wikipedia.org/wiki/Selection_sort) | [**Smoothsort**](https://en.wikipedia.org/wiki/Smoothsort) |
| **Best case** | \Theta(n) | \Theta(n \log n) | \Theta(n \log n) | \Theta(n \log n) | \Theta(n) | \Theta(n^2) | \Theta(n) |
| **Average case** | \Theta(n \log n) | \Theta(n \log n) | \Theta(n \log n) | \Theta(n \log n) | \Theta(n^2) | \Theta(n^2) | \Theta(n \log n) |
| **Worst case** | \Theta(n \log n) | \Theta(n \log n) | \Theta(n \log n) | \Theta(n^2) | \Theta(n^2) | \Theta(n^2) | \Theta(n \log n) |

**Time Complexity**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Timsort** | [**Merge sort**](https://en.wikipedia.org/wiki/Merge_sort) | [**Quicksort**](https://en.wikipedia.org/wiki/Quicksort) | [**Insertion sort**](https://en.wikipedia.org/wiki/Insertion_sort) | [**Selection sort**](https://en.wikipedia.org/wiki/Selection_sort) | [**Smoothsort**](https://en.wikipedia.org/wiki/Smoothsort) |
| **Space complexity** | O(n) | O(n) | O(\log n) | O(1) | O(1) | O(1) |

**Merge Sort**

**Time Complexity: Best, Worst, Average Case : O(n log n) , Space Complexity : O**

import java.util.Arrays;

**public** **class** MyMergeSort {

**public** **static** **void** mergeSort(**int**[] a) {

**if** (a.length <= 1) **return**;

**int**[] a1 = Arrays.*copyOfRange*(a, 0, a.length/2);

**int**[] a2 = Arrays.*copyOfRange*(a, a.length/2, a.length);

*mergeSort*(a1);

*mergeSort*(a2);

*merge*(a1, a2, a);

}

**private** **static** **void** merge(**int**[] a1, **int**[] a2, **int**[] a) {

**int** j = 0, k = 0;

**for** (**int** i = 0; i < a.length; i++) {

**if** (j == a1.length)

a[i] = a2[k++];

**else** **if** (k == a2.length)

a[i] = a1[j++];

**else** **if** (*less*(a1[j], a2[k]))

a[i] = a1[j++];

**else**

a[i] = a2[k++];

}

}

**public** **static** **boolean** less(**int** a , **int** b) {

**return** a < b ;

}

**public** **static** **void** main(String[] args) {

**int**[] a = {99,0,01,1,7,8,98,97,11,3,4,9,-1,};

*mergeSort*(a);

**for**( **int** i : a )

System.*out*.print(i+" ");

}

**<T> void merge(T[] a0, T[] a1, T[] a, Comparator<T> c) {**

**{**

**int i0 = 0, i1 = 0;**

**for (int i = 0; i < a.length; i++) {**

**if (i0 == a0.length)**

**a[i] = a1[i1++];**

**else if (i1 == a1.length)**

**a[i] = a0[i0++];**

**else if (compare(a0[i0], a1[i1]) < 0)**

**a[i] = a0[i0++];**

**else**

**a[i] = a1[i1++];**

**}**

**}**

}

**Generic Mergesort**

**<T> void mergeSort(T[] a, Comparator<T> c)**

**{**

**if (a.length <= 1) return;**

**T[] a0 = Arrays.copyOfRange(a, 0, a.length/2);**

**T[] a1 = Arrays.copyOfRange(a, a.length/2, a.length);**

**mergeSort(a0, c);**

**mergeSort(a1, c);**

**merge(a0, a1, a, c);**

**}**

**Merge two already sorted arrays sot that the resulting array, ie third array will be sorted one**

**The techniques used here is merge of Merge Sort**

**public class** MergeTwoSortedArrays1 {  
  
 **public static void** mergeTwoArrays(**int**[] a1 , **int**[] a2 , **int**[] a) {  
  
 **int j = 0 ;  
 int k = 0;  
  
 for (int i = 0; i < a.length; i++) {  
  
 if( j == a1.length )  
 a[i] = a2[k++];  
 else if( k == a2.length )  
 a[i] = a1[j++];  
 else if( *isLess*( a1[j] , a2[k]) )  
 a[i] = a1[j++];  
 else  
 a[i] = a2[k++];  
 }**  
 }  
  
 **public static boolean** isLess( **int** a , **int** b ) {  
 **return** a < b;  
 }  
  
 **public static void** main(String[] args) {  
  
 **int**[] a1 = {1,2,3,4,5,5};  
 **int**[] a2 = {7, 8, 9, 10, 14, 17};  
 **int**[] a3 = **new int**[a1.**length** + a2.**length**];  
  
 *mergeTwoArrays*(a1, a2, a3 );  
 **for**( **int** i : a3 )  
 System.***out***.print(i+**"\t"**);  
 }  
}

*//****An alternative technique to merge two arrays***

*//Super technique***public static void** merge(**int**[] a, **int**[] b, **int**[] c) {  
  
 **int** i = a.**length** - 1, j = b.**length** - 1, k = c.**length**;  
  
 **while** (k > 0)  
 c[--k] = (j < 0 || (i >= 0 && a[i] >= b[j])) ? a[i--] : b[j--];  
}

**Merge two arrays one is small and other big enough to accommodate the small one, the resulting array will be sorted**

**public class** MergeTwoArraysOneSmallOneBig {  
  
 **public static void** mergeArrays(**int**[] a, **int**[] b) {  
 *// pointer to end of the first sorted array* **int** i = a.**length**-1;  
 *// pointer to end of the second sorted array (pointer at 100 in below array b)* **int** j = b.**length**-a.**length**-1;  
 *// pointer to end of the second sorted array (pointer at last 0)* **int** k = b.**length**-1;  
  
 */\*\*  
 \* whichever is higher in two arrays, place that  
 \* element in last position of the bigger array  
 \*/* **while** ( i>=0 && j>=0 ) {  
 **if**( a[i] > b[j] ) {  
 b[k--] = a[i--];  
 } **else** {  
 b[k--] = b[j--];  
 }  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] aArr = {3,7,12,16};  
 **int**[] bArr = {1,4,9,100,0,0,0,0};  
  
 *mergeArrays*(aArr, bArr);  
  
 **for**(**int** i: bArr) {  
 System.***out***.println(i);  
 }  
 }  
}

**Merge Two Sorted Arrays, one is small and another is big enough to accommodate the small one.**  
  
**public class** TempMerger1 {  
 */\*\*  
 \** ***@param a*** *\** ***@param b,*** *b is the big array which cn accommodate a  
 \*/* **public static void** merge1(**int**[] a, **int**[] b) {  
  
 **int** i = a.**length** - 1;  
 **int** j = b.**length** - a.**length** - 1;  
 **int** k = b.**length** - 1;  
  
 **while** (i >= 0 && j >= 0 && k >= 0) {  
 **if** (a[i] > b[j])  
 b[k--] = a[i--];  
  
 **else** b[k--] = b[j--];  
 }  
  
 *//Check the left over from small array* **while** (i >= 0)  
 b[k--] = a[i--];  
  
 *//Check the left over from the big array* **while** (j >= 0)  
 b[k--] = b[j--];  
 }  
  
 **public static void** merge2(**int**[] a , **int**[] b) {  
 *//b[] is bigger* **int** j = 0 ;  
  
 **for** (**int** i = 0; i < b.**length** && j < a.**length**; i++) {  
 **if**( b[i] == 0 )  
 b[i] = a[j++];  
 }  
 *insertionSort*(b);;  
 }  
  
 **public static void** insertionSort( **int**[] a ) {  
 **for**( **int** i = 0 ; i < a.**length** ; i++ ) {  
 **for**( **int** j = i ; j > 0 && a[j] < a[j-1] ; j--)  
 *swap*(a, j , j-1);  
 }  
 }  
  
 **private static void** swap(**int**[] a , **int** i , **int** j ) {  
 **int** temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {1, 2, 5, 8, 9};  
*// int[] b = {3,4,5,11,34,0,0,0,0,0};* **int**[] b = {0,0,0,0,0,3,4,5,11,34}; *//Now it works  
  
// merge1(a,b);  
  
 merge2*(a,b);  
  
 **for**( **int** i : b )  
 System.***out***.print(i+**"\t"**);  
  
 }  
}

**Insert an element into an already sorted array**

**import** java.util.Arrays;  
  
**public class** InsertElementIntoSortedArray1 {  
  
 **public static void** insertElement(**int**[] a, **int** value){  
 *//find insert point* **int** i = 0;  
 *//In case of descending order while(i < a.length && a[i] > value)* **while**(i < a.**length** && a[i] < value) {  
 i++;  
 }  
 System.***out***.println(**"Value of I-->"**+i);  
 **if**(i < a.**length**) {  
 *//you found a place to insert the score* **for**(**int** j = a.**length**-1; j > i; j--){  
 a[j] = a[j - 1];  
 }  
 a[i] = value;  
 }  
 }  
  
 **public static void** main(String[] args) {  
  
 **int**[] a = {1,2,3,4,6,7};  
 **int**[] b = Arrays.*copyOf*(a, a.**length**+1);  
 *insertElement*(b,5);  
  
 **for**( **int** i : b )  
 System.***out***.print(i + **"\t"**);  
 }  
}

**Alternative way**

**public static int**[] insert(**int**[] a, **int** value) {  
 **int** index = Arrays.*binarySearch*(a, value);  
 System.***out***.println(**"Index ::: "** + index);  
 **int** newIndex = 0;  
 **if** (index < 0)  
 newIndex = -(index) - 1;  
 **else** newIndex = index + 1;  
 a = *insertElement*(a, value, newIndex);  
 **return** a;  
}  
  
**public static int**[] insertElement(**int**[] a, **int** value, **int** index) {  
 **int** length = a.**length**;  
 **int**[] destn = **new int**[length + 1];  
 System.*arraycopy*(a, 0, destn, 0, index);  
 destn[index] = value;  
 System.*arraycopy*(a, index, destn, index + 1, length - index);  
 **return** destn;  
}

**Insertion Sort**

**Best Case : O(n)**

**Worst Case and Average Case : O(*n*2)**

**The best case input is an array that is already sorted.**

**Why Insertion Sort is used in Java ?**

<http://stackoverflow.com/questions/3566843/why-does-java-util-arrays-sortobject-use-2-kinds-of-sorting-algorithms>

It's important to note that an algorithm that is O(N log N) is not always faster in practice than an O(N^2) algorithm. It depends on the constants, and the range of N involved. (Remember that[asymptotic notation](http://en.wikipedia.org/wiki/Big_O_notation) measures relative growth rate, not absolute speed).

For small N, insertion sort in fact does beat merge sort. It's also faster for almost-sorted arrays.

Here's [a quote](http://www.sorting-algorithms.com/insertion-sort):

Although it is one of the elementary sorting algorithms with O(N^2) worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead).

For these reasons, and because it is also stable, insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

straight insertion sort is best for small or very nearly sorted lists

What this means is that, in practice:

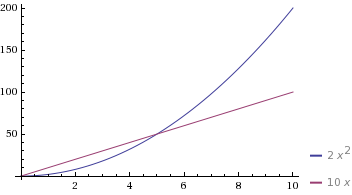
* Some algorithm A1 with higher asymptotic upper bound may be preferable than another known algorithm A2 with lower asymptotic upper bound
  + Perhaps A2 is just too complicated to implement
  + Or perhaps it doesn't matter in the range of N considered
    - See e.g. [Coppersmith–Winograd algorithm](http://en.wikipedia.org/wiki/Coppersmith%E2%80%93Winograd_algorithm)
* Some hybrid algorithms may adapt different algorithms depending on the input size

### A numerical example

Let's consider these two functions:

* f(x) = 2x^2; this function has a quadratic growth rate, i.e. "O(N^2)"
* g(x) = 10x; this function has a linear growth rate, i.e. "O(N)"

Now let's plot the two functions together:

  
Source:[WolframAlpha: plot 2x^2 and 10x for x from 0 to 10](http://www.wolframalpha.com/input/?i=plot+2x%5E2+and+10x+for+x+from+0+to+10)

Note that between x=0..5, f(x) <= g(x), but for any larger x, f(x) quickly outgrows g(x).

Analogously, if A1 is a quadratic algorithm with a low overhead, and A2 is a linear algorithm with a high overhead, for smaller input, A1 may be faster than A2.

Thus, you can, should you choose to do so, create a hybrid algorithm A3 which simply selects one of the two algorithms depending on the size of the input. Whether or not this is worth the effort depends on the actual parameters involved.

Many tests and comparisons of sorting algorithms have been made, and it was decided that because insertion sort beats merge sort for small arrays, it was worth it to implement both for [Arrays.sort](http://download.oracle.com/javase/6/docs/api/java/util/Arrays.html#sort%28java.lang.Object%5b%5d%29).

It's for speed. The overhead of mergeSort is high enough that for short arrays it would be slower than insertion sort.

[Insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) works well on small or nearly sorted lists.

<http://stackoverflow.com/questions/736920/is-there-ever-a-good-reason-to-use-insertion-sort>

An important concept in analysis of algorithms is **asymptotic** analysis. In the case of two algorithms with different asymptotic running times, such as one O(n^2) and one O(nlogn) as is the case with insertion sort and quicksort respectively, **it is not definite that one is faster than the other.**

The important distinction with this sort of analysis is that for **sufficiently large N**, one algorithm will be faster than another. When analyzing an algorithm down to a term like O(nlogn), you drop constants. When realistically analyzing the running of an algorithm, those constants will be important only for situations of small n.

So what does this mean? That means for certain small n, some algorithms are faster. This [article](http://embeddedgurus.com/stack-overflow/2009/03/sorting-in-embedded-systems) from EmbeddedGurus.net includes an interesting perspective on choosing different sorting algorithms in the case of a limited space (16k) and limited memory system. Of course, the article references only sorting a list of 20 integers, so larger orders of n is irrelevant. Shorter code and less memory consumption (as well as avoiding recursion) were ultimately more important decisions.

Insertion sort has low overhead, it can be written fairly succinctly, and it has several two key benefits: it is stable, and it has a fairly fast running case when the input is nearly sorted.

**Insertion Sort Code**

**public class** InsertionSort {  
  
 **private static void** insertionSort(**int** x[], **int** off, **int** len) {  
*// Insertion sort on smallest arrays* **for** (**int** i = off; i < len + off; i++)  
 **for** (**int** j = i; j > off && x[j - 1] > x[j]; j--)  
 *swap*(x, j, j - 1);  
 }  
  
 **private static void** insertionSort1(**int** a[]) *//Insertion sort on small arrays* {  
 **for** (**int** i = 0; i < a.**length**; i++) {  
 **for** (**int** j = i; j > 0 && a[j] < a[j - 1]; j--)  
 *swap*(a, j, j - 1);  
 }  
 }  
  
 **private static void** swap(**int**[] a, **int** i, **int** j) {  
 **int** temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] a = {-1, 67, 8, -1, 1, 3, 4, 10, 05, 07, 9};  
*// int[] a = { 67, 89, -1, 2, 3, 99, 23, 05, 07, 9,1};* **for** (**int** k : a)  
 System.***out***.print(k + **" "**);  
 System.***out***.println(**"\n"**);  
*// insertionSort(a, 0, a.length);  
 insertionSort1*(a);  
 **for** (**int** k : a)  
 System.***out***.print(k + **" "**);  
 }  
  
}

**Selection sort**

In computer science, a selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n2) time complexity, making it inefficient on large lists. It works on the principle that : first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

**Best case** : O(n). It occurs when the data is in sorted order. After making one pass through the data and making no insertions, insertion sort exits.

**Average case**: O(n2) since there is a wide variation with the running time.

**Worst case** : O(n2 ) if the numbers were sorted in reverse order. The worst case occurs if the array is already sorted in descending order

public class SelectionSort

{

public static void selectionSort( int[] a )

{

for( int i = 0 ; i < a.length ; i++ )

{

int minIndex = i;

for( int j = i ; j < a.length ; j++ )

{

if( a[j] < a[minIndex])

minIndex = j;

}

//swap between i and minIndex

int temp = a[i];

a[i] = a[minIndex];

a[minIndex] = temp;

}

}

public static void main(String[] args)

{

// int[] a = {-1, 67, 8, -1, 1, 3, 4, 10, 05, 07, 9};

int[] a = { 67, 89, -1, 2, 3, 99, 23, 05, 07, 9,1};

for( int k : a )

System.*out*.print(k+" ");

System.*out*.println("\n");

// insertionSort(a, 0, a.length);

*selectionSort*(a);

for( int k : a )

System.*out*.print(k+" ");

}

}

**Java Data Structure**

# **ArrayList**

**import** java.util.Arrays;  
**public class** MyArrayListList {  
 **private int size**;  
 **private int defaultCapacity** = 10;  
 **private** Object[] **elements**;  
  
 **public** MyList() {  
 **elements** = **new** Object[**defaultCapacity**];  
 }  
  
 **public void** add(Object x) {  
 **if**( **elements**.**length** == **size** )  
 ensureCapacity();  
 **elements**[**size**++] = x;  
 }  
  
 **private void** ensureCapacity() {  
 **elements** = Arrays.*copyOf*(**elements**, (**elements**.**length**\*2) );  
 }  
  
 **public** Object get( **int** index ) {  
 **return elements**[index];  
 }  
  
 **public void** delete( Object obj ) {  
 **for**( **int** i = 0 ; i < **elements**.**length** ; i++ ) {  
 **if**( **elements**[i].equals(obj)) {  
 **elements**[i] = **null**;  
 **size**--;  
 condenseArray(i);  
 **break**;  
 }  
 }  
 }  
  
 **public int** size() {  
 **return size**;  
 }  
  
 **private void** condenseArray( **int** start ) {  
 **for**( **int** i = start ; i < **size** ; i++ )  
 **elements**[i] = **elements**[i+1];  
 }  
  
 **public void** trimArray() {  
 **elements** = Arrays.*copyOf*(**elements**, **size**);  
 }

**public static void** main(String[] args)  
 {  
 MyArrayListList list = **new** MyArrayListList ();  
  
 **for**( **int** i = 0 ; i < 13 ; i++ )  
 {  
 list.add( **new** Integer(i));  
 }  
  
 System.***out***.println(**"Size ::: "**+list.size());  
  
 list.delete( **new** Integer(5));  
 System.***out***.println(**"Size ::: "**+list.size());  
 list.delete( **new** Integer(7));  
 System.***out***.println(**"Size ::: "**+list.size());  
 **for**( **int** i = 0 ; i < list.size() ; i++ )  
 {  
 System.***out***.println(list.get(i));  
 }  
 }  
}

**Tree Stucture in Java**

In case of tree structure we can get the child information and its parent. Sometimes interviewer ask questions like how will design a structure so that for a particular employee, we can get its Manager and its children. Interviewer will ask the question that how will you design your object model to accommodate company hierarchy. Example, Managing Director or President of the Company has CEO,CTO,COO, each CEO has Delivery Head, each Delivery Head has List of Project managers etc. At any point of time you can get the Parent and the Children.

**import** java.util.List;

**import** java.util.ArrayList;

**public** **class** TreeNode

{

**private** TreeNode parent = **null**;

**private** List children = **null**;

**private** Object reference;

**public** TreeNode(Object obj) {

**this**.parent = **null**;

**this**.reference = obj;

**this**.children = **new** ArrayList();

}

**public** **void** addChildNode(TreeNode child) {

child.parent = **this**;

**if** (!children.contains(child))

children.add(child);

}

**public** List getChildren() {

**return** children;

}

**public** TreeNode getParent() {

**return** parent;

}

**public** String toString() {

**return** reference.toString();

}

**public** **static** **void** main(String[] args)

{

TreeNode root = **new** TreeNode( "Root" );

TreeNode child1 = **new** TreeNode("Child1");

root.addChildNode(child1);

TreeNode child2 = **new** TreeNode("Child2");

root.addChildNode(child2);

TreeNode grandChild1 = **new** TreeNode("GrandChild1");

child1.addChildNode(grandChild1);

TreeNode grandChild2 = **new** TreeNode("GrandChild2");

child2.addChildNode(grandChild2);

List list = root.getChildren();

**for**( **int** i = 0 ; i < list.size() ; i++ ) {

TreeNode tt = (TreeNode) list.get(i);

System.*out*.println(tt+"----"+tt.getParent());

List list1 = tt.getChildren();

**for**( **int** j = 0 ; j < list1.size() ; j++ ) {

TreeNode grand = (TreeNode) list1.get(j);

System.*out*.println(grand+"----"+grand.getParent());

}

}

}

}

**Another example of similar kind is given below.**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** EmpNode

{

**private** EmpNode parentEmp;

**private** List childrenEmps;

**private** String name;

**private** String id;

**public** EmpNode( String id , String name )

{

**this**.parentEmp = **null**;

**this**.name = name;

**this**.id = id;

**this**.childrenEmps = **new** ArrayList();

}

**public** EmpNode getParent()

{

**return** parentEmp;

}

**public** List getChildren()

{

**return** childrenEmps;

}

**public** **void** addEmp( EmpNode childEmp )

{

childEmp.parentEmp = **this**;

**if**( !childrenEmps.contains(childEmp) )

childrenEmps.add(childEmp);

}

**public** String toString()

{

**return** id+"---"+name;

}

**public** **static** **void** main(String[] args)

{

EmpNode root = **new** EmpNode( "1","Root" );

EmpNode child1 = **new** EmpNode("2","Child1");

root.addEmp(child1);

EmpNode child2 = **new** EmpNode("3","Child2");

root.addEmp(child2);

EmpNode grandChild1 = **new** EmpNode("4","GrandChild1");

child1.addEmp(grandChild1);

EmpNode grandChild2 = **new** EmpNode("5","GrandChild2");

child2.addEmp(grandChild2);

List list = root.getChildren();

**for**( **int** i = 0 ; i < list.size() ; i++ ) {

EmpNode tt = (EmpNode) list.get(i);

System.*out*.println(tt+"----"+tt.getParent());

List list1 = tt.getChildren();

**for**( **int** j = 0 ; j < list1.size() ; j++ ) {

EmpNode grand = (EmpNode) list1.get(j);

System.*out*.println(grand+"----"+grand.getParent());

}

}

}

}

**LinkedList in Java**

**LinkedList in Java**

//Reference http://www.cs.bu.edu/fac/snyder/cs112/CourseMaterials/LinkedListNotes.html

**public** **class** LinkedList1

{

**private** Node front;

**private** Node back;

**private** **static** **class** Node

{

**private** Object element;

**private** Node next;

**public** Node( Object data )

{

element = data;

next = **null**;

}

}

**private** **boolean** isEmpty()

{

**return** front == **null**;

}

**public** **void** add(Object x)

{

**if**( isEmpty() )

back = front = **new** Node(x);

**else**

back = back.next = **new** Node(x);

}

**public** **void** addFirst(Object x) {

Node f = front;

front = **new** Node(x);

front.next = f;

}

**public** Object get( **int** index )

{

**int** counter = 0;

Object data = **null**;

**for**( Node p = front ; p != **null** ; p = p.next )

{

**if**( counter == index )

{

data = p.element;

**break**;

}

counter++;

}

**return** data;

}

//Print all the elements

**public** **void** print()

{

**for**( Node p = front ; p != **null** ; p = p.next )

System.*out*.print(p.element+" ");

}

**public** **void** print() {

**printRecursive(front);**

}

**public void printRecursive(Node p) {**

**if( p != null) {**

System.*out*.print(p.element+" ");

**printRecursive(p.next)**

**}**

**}**

// Print in reverse

**public** **void** printReverseList()

{

printReverseList(front);

}

**private** **void** printReverseList( Node p )

{

**if** (p != **null**)

{

printReverseList( p.next );

System.*out*.println(p.element);

}

}

//~~~~~~~~~~~~~~~~~~~~ size() or length() operations ~~~~~~~~~~~~~~~~~~~~~~~~~~

**public** **int** size()

{

**int** counter = 0;

**for**(Node p = front ; p != **null**; p = p.next )

counter++;

**return** counter;

}

**public** **int** length()

{

**return** length(front);

}

**public** **int** length( Node p )

{

// Example call: int len = length(head.next);

**if** (p == **null**)

**return** 0;

**else**

**return** 1 + length( p.next );

}

//for( Node p = front.next,q = front; p != null ; q = p , p = p.next )

// During first iteration, p points to first element and q to the header node;

// thereafter, p points to a node and q points to the previous node

**public** **void** deleteObject(Object x)

{

**if** ( front != **null** && front.element.equals(x) )

{ // special case, have to delete first node?

front = front.next;

}

**else**

{

**for**( Node p = front,q = front; p != **null** ; q = p , p = p.next )

{

**if**( p.element.equals(x) )

{

q.next = p.next;

**break**;

}

}

}

}

**public** Object deleteFirst() //POLL operation

{

**if**(isEmpty()) **throw** **new** RuntimeException("No data");

Object data = front.element;

front = front.next;

**return** data;

}

**public** Object deleteLast( ) //POP() operation

{

**if**( isEmpty( ) )

**throw** **new** RuntimeException( "No data" );

Object returnValue = back.element;

**if**( front == back )

front = back = **null**;

**else**

{

Node current = front;

**while** ( current.next != back ) // not last node

current = current.next; // move to next node

back = current;

current.next = **null**;

}

**return** returnValue;

}

//Another way of writing delete last

**public** Object deleteLast1( ) //POP() operation

{

**if**( front == **null** ) **throw** **new** RuntimeException( "No data" );

Object returnValue = back.element;

**if**( front == back ) front = back = **null**;

**else**

{

Node p = front;

**for**( ; p.next != back ; p = p.next ) {

//Nothing to do

}

back = p;

p.next = **null**;

}

**return** returnValue;

}

**public** Object pop()

{

**return** deleteLast();

}

**public** Object poll()

{

**return** deleteFirst();

}

**public** **void** deleteAt(**int** p)

{

front = deleteAt(p, front);

}

**private** Node deleteAt(**int** p, Node node)

{

**if**(node == **null**)

**throw** **new** java.util.NoSuchElementException("cannot delete.");

**else**

**if**(p == 0)

**return** node.next;

**else**

node.next = deleteAt(p-1, node.next);

**return** node;

}

//Find the middle element

**public** **void** findMiddle() {

**int** length = 0;

Node middle = front;

**for**( Node p = front ; p != **null** ; p = p.next ) {

length++;

**if**( length % 2 == 0 )

middle = middle.next;

}

System.*out*.println("Middle Element : "+middle.element);

}

//Reverse the LinkedList

**public** **void** reverseList() {

recursiveReverse(front);

}

**public** **void** reverse(Node h)

{

Node p = h.next, q = h, r;

**while** ( p != **null** ) {

r = q; // r follows q

q = p; // q follows p

p = p.next; // p moves to next node

q.next = r; // link q to preceding node

}

front.next.next = **null**;

front.next = q;

}

**public** **void** recursiveReverse(Node currentNode )

{

//check for empty list

**if**(currentNode == **null**)

**return**;

/\* if we are at the TAIL node:

recursive base case:

\*/

**if**(currentNode.next == **null**)

{

//set HEAD to current TAIL since we are reversing list

front = currentNode;

**return**; //since this is the base case

}

recursiveReverse(currentNode.next);

currentNode.next.next = currentNode;

currentNode.next = **null**; //set "old" next pointer to NULL

}

**public** **static** **void** main(String[] args) {

LinkedList1 list = **new** LinkedList1();

//Add elements

**for**( **int** i = 0 ; i < 17 ; i++ )

list.add(**new** Integer(i));

//Display elements

list.print();

System.*out*.println("\n");

//Display First Time

System.*out*.println("-------------Display First Time----------");

**for**( **int** i = 0 ; i < list.size() ; i++ )

System.*out*.print(list.get(i)+" ");

//Display Second Time

System.*out*.println("\n-------------Display Second Time----------");

**for**( **int** i = 0 ; i < list.size() ; i++ )

System.*out*.print(list.get(i)+" ");

list.deleteFirst();//First object

list.deleteLast();//Last Object

System.*out*.println("\n");

list.print();

System.*out*.println("\n");

//POP() operation

System.*out*.println("pop() : "+list.pop());

System.*out*.println("pop() : "+list.pop());

System.*out*.println("\n");

list.print();

//POLL() operation

System.*out*.println("poll() : "+list.poll());

System.*out*.println("poll() : "+list.poll());

list.print();

//Delete from the index

System.*out*.println("\nDelete from the middle");

list.deleteAt(8);

list.print();

System.*out*.println("\nDelete an object");

list.deleteObject(**new** Integer(13));

list.print();

System.*out*.println("\n");

list.findMiddle();

System.*out*.println("All elements .....");

list.print();

list.reverseList();

System.*out*.println("\n");

list.print();

}

}

Detect Loop Inside a LinkedList

public class LoopInLinkedList {

private Node front;

private Node back;

private static class Node {

private Node next;

private Object element;

public Node(Object data) {

this.element = data;

this.next = null;

}

}

public void addLast(Object x) {

if (front == null)

back = front = new Node(x);

else

back = back.next = new Node(x);

}

**//Floyd’s Cycle Detection Algorithm Tortoise and Hare**

**public boolean isCyclic() {**

**Node fast = front;**

**Node slow = front;**

**while (fast != null && fast.next != null) {**

**fast = fast.next.next;**

**slow = slow.next;**

**//if fast and slow pointers are meeting then LinkedList is cyclic**

**if (fast == slow) {**

**return true;**

**}**

**}**

**return false;**

**}**

public void print() {

for (Node p = front; p != null; p = p.next) {

System.out.print(p.element + "\t");

}

}

**//Method to create a Loop inside a LinkedList**

**public void createALoop() {**

**front.next.next.next.next.next.next = front.next.next;**

**}**

public static void main(String[] args) {

LoopInLinkedList sl = new LoopInLinkedList();

for (int i = 0; i < 10; i++)

sl.addLast(new Integer(i));

sl.print();

System.out.println("\nIs SL Is cyclic :::" + sl.isCyclic());

sl.createALoop();

System.out.println("\nIs SL Is cyclic :::" + sl.isCyclic());

}

}

**A simple LinkedList**

**public** **class** MyLinkedList

{

// reference to the head node.

**private** Node head;

**private** **int** listCount;

**private** **class** Node

{

// reference to the next node in the chain,

**private** Node next;

// data carried by this node.

**private** Object data;

**public** Node(Object data)

{

**this**.next = **null**;

**this**.data = data;

}

}

**public** MyLinkedList()

{

// this is an empty list, so the reference to the head node

// is set to a new node with no data

head = **new** Node(**null**);

listCount = 0;

}

**public** **void** add(Object data)

{

// post: appends the specified element to the end of this list.

Node temp = **new** Node(data);

Node currentNode = head;

// starting at the head node, crawl to the end of the list

**while**(currentNode.next != **null**)

{

currentNode = currentNode.next;

}

// the last node's "next" reference set to our new node

currentNode.next = temp;

listCount++;// increment the number of elements variable

}

**public** **void** add1( Object x ) {

**if**( head == **null** ) head = **new** Node(x);

**else**

{

Node current = head;

**for**( Node p = current ; p != **null** ; p = p.next ) {

current = p;

}

current.next = **new** Node(x);

}

listCount++;// increment the number of elements variable

}

**public** **int** size()

{

**return** listCount;

}

**public** Object get(**int** index)

{

// post: returns the element at the specified position in this list.

// index must be 1 or higher

**if**(index < 0)

**return** **null**;

Node current = head.next;

**for**(**int** i = 0; i < index; i++)

{

**if**(current.next == **null**)

**return** **null**;

current = current.next;

}

**return** current.data;

}

**public** **boolean** remove(**int** index) {

// post: removes the element at the specified position in this list.

// if the index is out of range, exit

**if**(index < 0 || index > size())

**return** **false**;

Node currentNode = head;

**for**(**int** i = 0; i < index; i++) {

**if**(currentNode.next == **null**)

**return** **false**;

currentNode = currentNode.next;

}

currentNode.next = currentNode.next.next;

listCount--; // decrement the number of elements variable

**return** **true**;

}

**public** String toString()

{

Node current = head.next;

String output = "";

**while**(current != **null**)

{

output += "[" + current.data.toString() + "]";

current = current.next;

}

**return** output;

}

**public** **static** **void** main(String[] args)

{

MyLinkedList list = **new** MyLinkedList();

list.add1("abcd");

list.add1("bcd");

list.add1("pqrs");

System.*out*.println(list);

**for**( **int** i = 0 ; i < list.size() ; i++ )

System.*out*.print(list.get(i)+" ");

list.remove(1);

System.*out*.println("\n"+list);

}

}

**Binary Search**

In computer science, a **binary search or half-interval search algorithm** finds the position of a specified input value (the search "key") within an array sorted by key value.In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. A binary search is a dichotomic **divide and conquer search algorithm**.

**Worst case performance O(log n)**

**Best case performance O(1)**

**Average case performance O(log n)**

**Worst case space complexity O(1)**

**If we had a sorted list of 137 elements would the:**

**Worst case complexity be O=log2(n) = 8**

**Average case complexity be O=log(n) = 2**

**Best case complexity be O=n/2=69**

Best case is O(1). You could get lucky and the element you want is in the middle of the list.

Worst case is O(log2(n)) as the number of times you can divide the list up in 2 is the maximum times you'll have to compare elements in a binary search.

Average case is also O(log2(n)). The average number of times you would compare elements in a binary search is halfway between 1 and log2(n), so it's 0.5\*log2(n). But the "0.5\*" is considered insignificant compared to the rest, so it ends up being O(log2(n)) as well.

**Best case - O (1) comparisons**

**In the best case, the item X is the middle in the array A. A constant number of comparisions (actually just 1) are required.**

**Worst case - O (log n) comparsions**

**In the worst case, the item X does not exist in the array A at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling(lg n ) times. Thus, ceiling(lg n ) comparisons are required.**

**Average case - O (log n) comparsions**

**To find the average case, take the sum over all elements of the product of number of comparsions required to find each element and the probability of searching for that element. To simplify the analysis, assume that no item which is not in A will be searched for, and that the probabilities of searching for each element are uniform**.

**Binary search code is given below.**

**public** **class** BinarySearch {

**public** **static** **boolean** binarySearch( **int**[] a , **int** b ) {

**int** low = 0;

**int** high = a.length - 1;

**while**( low <= high ) {

**int** mid = low + (high - low) / 2;

**if**( b < a[mid] ) high = mid - 1;

**else** **if**( b > a[mid] ) low = mid+1;

**else** **return** **true**;

}

**return** **false**;

}

**public** **static** **void** main(String[] args) {

**int**[] a = {1,5,9,11,22,43,77};

System.*out*.println("Binary Search found : " + *binarySearch*(a, 11));

System.*out*.println("Binary Search found : " + *binarySearch*(a, 31));

}

}

**public** **class** BinarySearch1 {

**public** **static** **int** binarySearch( **int**[] a , **int** b ) {

**int** low = 0;

**int** high = a.length - 1;

**while**( low <= high ) {

**int** mid = low + (high - low) / 2;

**if**( b < a[mid] ) high = mid - 1;

**else** **if**( b > a[mid] ) low = mid+1;

**else** **return** mid;

}

**return** -(low+1);

}

**public** **static** **void** main(String[] args) {

**int**[] a = {1,5,9,11,22,43,77};

System.*out*.println("Binary Search found : " + *binarySearch*(a, 11));

System.*out*.println("Binary Search found : " + *binarySearch*(a, 31));

}

}

**Binary search tree**

From Wikipedia, the free encyclopedia : <http://en.wikipedia.org/wiki/Binary_search_tree>

|  |  |  |
| --- | --- | --- |
| Binary search tree  [http://upload.wikimedia.org/wikipedia/commons/thumb/d/da/Binary_search_tree.svg/200px-Binary_search_tree.svg.png](http://en.wikipedia.org/wiki/File:Binary_search_tree.svg) | | |
| [Type](http://en.wikipedia.org/wiki/List_of_data_structures) | [Tree](http://en.wikipedia.org/wiki/Tree_(data_structure)) | |
| [Time complexity](http://en.wikipedia.org/wiki/Time_complexity) in [big O notation](http://en.wikipedia.org/wiki/Big_O_notation) | | |
|  | Average | Worst case |
| Space | O(n) | O(n) |
| Search | O(log n) | O(n) |
| Insert | O(log n) | O(n) |
| Delete | O(log n) | O(n) |

A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

In [computer science](http://en.wikipedia.org/wiki/Computer_science), a **binary search tree** (**BST**), sometimes also called an **ordered** or **sorted binary tree**, is a [node](http://en.wikipedia.org/wiki/Node_(computer_science))-based [binary tree](http://en.wikipedia.org/wiki/Binary_tree) data structure which has the following properties:

* **The left**[**subtree**](http://en.wikipedia.org/wiki/Tree_(data_structure)#Subtree)**of a node contains only nodes with keys less than the node's key.**
* **The right subtree of a node contains only nodes with keys greater than the node's key.**
* **The left and right subtree each must also be a binary search tree.**
* **There must be no duplicate nodes**.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

The major advantage of binary search trees over other [data structures](http://en.wikipedia.org/wiki/Data_structure) is that the related [sorting algorithms](http://en.wikipedia.org/wiki/Sorting_algorithm) and [search algorithms](http://en.wikipedia.org/wiki/Search_algorithm) such as [in-order traversal](http://en.wikipedia.org/wiki/In-order_traversal) can be very efficient.

### Traversal

Once the binary search tree has been created, its elements can be retrieved [in-order](http://en.wikipedia.org/wiki/In-order_traversal) by [recursively](http://en.wikipedia.org/wiki/Recursion) traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as it's recursively accessed. As with all binary trees, one may conduct a [pre-order traversal](http://en.wikipedia.org/wiki/Pre-order_traversal) or a [post-order traversal](http://en.wikipedia.org/wiki/Post-order_traversal), but neither are likely to be useful for binary search trees. An in-order traversal of a binary search tree will always result in a sorted list of node items (numbers, strings or other comparable items).

### Depth-first

There are three types of depth-first traversal: pre-order,[[1]](http://en.wikipedia.org/wiki/Tree_traversal#cite_note-holtenotes-1) in-order,[[1]](http://en.wikipedia.org/wiki/Tree_traversal#cite_note-holtenotes-1) and post-order.[[1]](http://en.wikipedia.org/wiki/Tree_traversal#cite_note-holtenotes-1) For a binary tree, they are defined as operations recursively at each node, starting with the root node follows:

**Pre-order**

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

**In-order (symmetric)**

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

**Post-order**

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

The trace of a traversal is called a sequentialisation of the tree. No one sequentialisation according to pre-, in- or post-order describes the underlying tree uniquely. Given a tree with distinct elements, either pre-order or post-order paired with in-order is sufficient to describe the tree uniquely. However, pre-order with post-order leaves some ambiguity in the tree structure

#### **Pre-Order**

#### **In-Order**

void inorder(struct tree\_node \*p)

{ if (p !=NULL)

{

inorder(p->left\_child);

printf(“%d\n”, p->data);

inorder(p->right\_child);

}

}

void *preorder*(struct tree\_node \* p)

{ if (p !=NULL)

{

printf(“%d\n”, p->data);

preorder(p->left\_child);

preorder(p->right\_child);

}

}

***Post Order***

void postorder(struct tree\_node \*p)

{ if (p !=NULL)

{

postorder(p->left\_child);

postorder(p->right\_child);

printf(“%d\n”, p->data);

}

}

**Preorder Traversal**

the node is visited before its left and right subtrees,

**Postorder Traversal**

the node is visited after both subtrees.

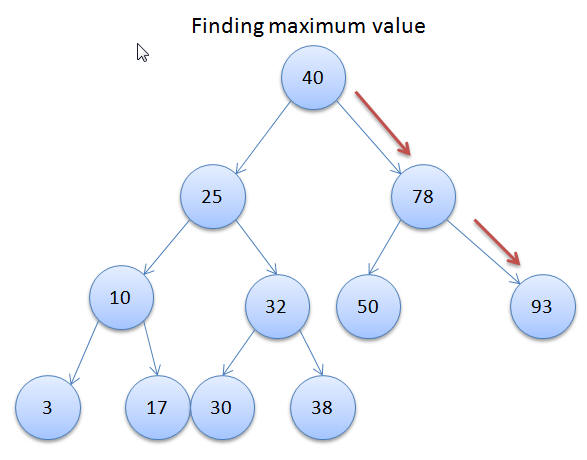
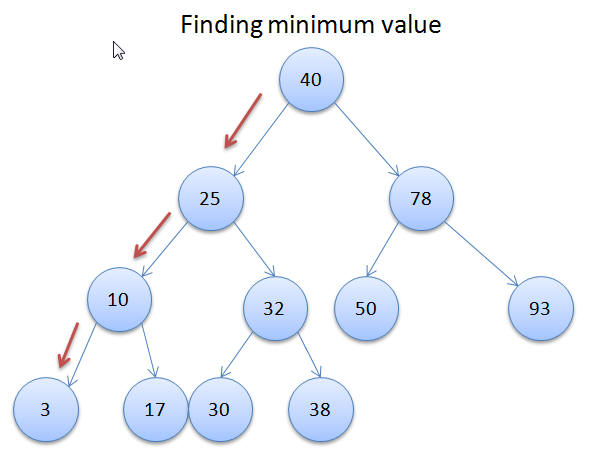
**Inorder Traversal**

the node is visited between the subtrees,

Visit left subtree, visit the node, and visit the right subtree.

* A binary tree is ***height balanced*** (or ***balanced***), if the height of any node’s right subtree differs from the height of the node’s left subtree by no more than 1.

**Perfect binary tree = a binary tree where each level contains the maximum number of nodes every level is completely full of nodes**

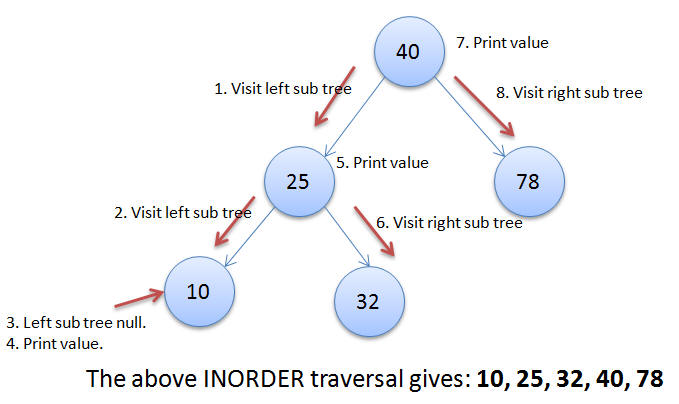
## Traversing the Binary Search Tree (BST)

Traversing the tree or BST in this case is visiting each of the nodes present in the tree and performing some operation with the value present in the node which in this case will be printing the value present in the node. **When we traverse the tree we have to visit the value present in the node, then node’s right sub tree and the left sub tree**. Visiting the right and left sub tree will be a recursive operation. The order in which we perform the three operations i.e visiting the value, right sub tree and left sub tree gives rise to three traversal techniques:

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

**Inorder Traversal**

In this traversal the left sub tree of the given node is visited first, then the value at the given node is printed and then the right sub tree of the given node is visited. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



**private void printInOrderRec(Node currRoot){**

**if ( currRoot == null ){**

**return;**

**}**

**printInOrderRec(currRoot.left);**

**System.out.print(currRoot.value+", ");**

**printInOrderRec(currRoot.right);**

**}**

**public void printInorder(){**

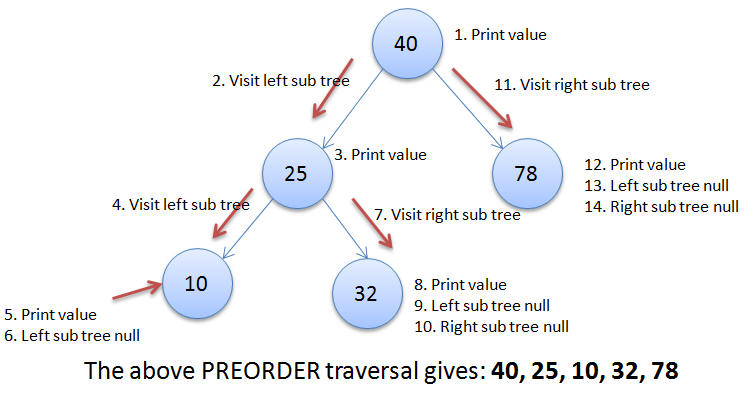
**printInOrderRec(root);**

**System.out.println("");**

**}**

**Preorder traversal**

In this traversal the value at the given node is printed first and then the left sub tree of the given node is visited and then the right sub tree of the given node is visited. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



public void printPreorder() {

printPreOrderRec(root);

System.out.println("");

}

private void printPreOrderRec(Node currRoot) {

if (currRoot == null) {

return;

}

System.out.print(currRoot.value + ", ");

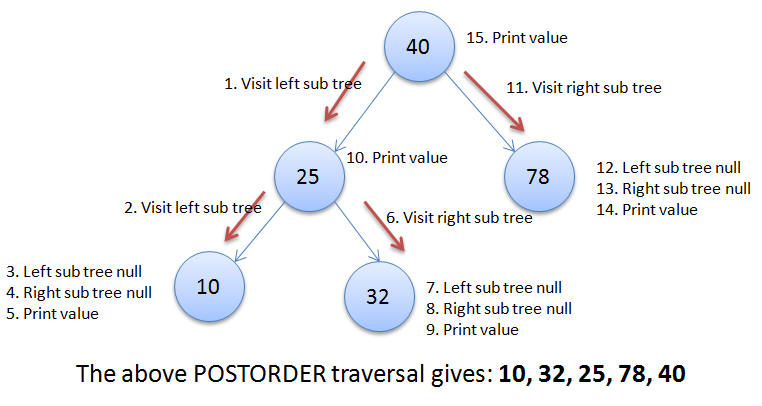
printPreOrderRec(currRoot.left);

printPreOrderRec(currRoot.right);

}

**Postorder Traversal**

In this traversal the left sub tree of the given node is traversed first, then the right sub tree of the given node is traversed and then the value at the given node is printed. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



**public void printPostorder() {**

private void printPostOrderRec(Node currRoot) {

if (currRoot == null) {

return;

}

printPostOrderRec(currRoot.left);

printPostOrderRec(currRoot.right);

System.out.print(currRoot.value + ", ");

}

**printPostOrderRec(root);**

**System.out.println("");**

**}**

Example code is given below ,

this code is of type Builder design pattern.

public class Node<T> {

//The value present in the node.

public int value;

//The reference to the left subtree.

public Node left;

//The reference to the right subtree.

public Node right;

public Node(int value) {

this.value = value;

}

}

/\*\*

\* Represents the Binary Search Tree.

\*/

public class BinarySearchTree {

//Refrence for the root of the tree.

public Node root;

public BinarySearchTree insert(int value) {

Node node = new Node<>(value);

if (root == null) {

root = node;

return this;

}

insertRec(root, node);

return this;

}

private void insertRec(Node latestRoot, Node node) {

if (latestRoot.value > node.value) {

if (latestRoot.left == null) {

latestRoot.left = node;

return;

} else {

insertRec(latestRoot.left, node);

}

} else {

if (latestRoot.right == null) {

latestRoot.right = node;

return;

} else {

insertRec(latestRoot.right, node);

}

}

}

/\*\*

\* Returns the minimum value in the Binary Search Tree.

\*/

public int findMinimum() {

if (root == null) {

return 0;

}

Node currNode = root;

while (currNode.left != null) {

currNode = currNode.left;

}

return currNode.value;

}

/\*\*

\* Returns the maximum value in the Binary Search Tree

\*/

public int findMaximum() {

if (root == null) {

return 0;

}

Node currNode = root;

while (currNode.right != null) {

currNode = currNode.right;

}

return currNode.value;

}

/\*\*

\* Printing the contents of the tree in an inorder way.

\*/

public void printInorder() {

printInOrderRec(root);

System.out.println("");

}

/\*\*

\* Helper method to recursively print the contents in an inorder way

\*/

private void printInOrderRec(Node currRoot) {

if (currRoot == null) {

return;

}

printInOrderRec(currRoot.left);

System.out.print(currRoot.value + ", ");

printInOrderRec(currRoot.right);

}

/\*\*

\* Printing the contents of the tree in a Preorder way.

\*/

public void printPreorder() {

printPreOrderRec(root);

System.out.println("");

}

/\*\*

\* Helper method to recursively print the contents in a Preorder way

\*/

private void printPreOrderRec(Node currRoot) {

if (currRoot == null) {

return;

}

System.out.print(currRoot.value + ", ");

printPreOrderRec(currRoot.left);

printPreOrderRec(currRoot.right);

}

/\*\*

\* Printing the contents of the tree in a Postorder way.

\*/

public void printPostorder() {

printPostOrderRec(root);

System.out.println("");

}

/\*\*

\* Helper method to recursively print the contents in a Postorder way

\*/

private void printPostOrderRec(Node currRoot) {

if (currRoot == null) {

return;

}

printPostOrderRec(currRoot.left);

printPostOrderRec(currRoot.right);

System.out.print(currRoot.value + ", ");

}

}

public class BinarySearchTreeDemo {

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

bst .insert(40)

.insert(25)

.insert(78)

.insert(10)

.insert(3)

.insert(17)

.insert(32)

.insert(30)

.insert(38)

.insert(78)

.insert(50)

.insert(93);

System.out.println("Inorder traversal");

bst.printInorder();

System.out.println("Preorder Traversal");

bst.printPreorder();

System.out.println("Postorder Traversal");

bst.printPostorder();

System.out.println("The minimum value in the BST: " + bst.findMinimum());

System.out.println("The maximum value in the BST: " + bst.findMaximum());

}

}

A **balanced binary tree** is commonly defined as a binary tree in which the depth of the left and right subtrees of every node differ by 1 or less,

The depth of a node M in a tree is the length of the path from the root of the tree to M. The height of a tree is one more than the depth of the deepest node in the tree. All nodes of depth d are at level d in the tree. The root is the only node at level 0, and its depth is 0.

**HEIGHT** is defined as the number of nodes in the longest path from the root node to a leaf node. Therefore: a tree with only a root node has a height of 1 and not 0.

The **LEVEL** of a given node is the distance from the root plus 1. Therefore: The root is on level 1, its child nodes are on level 2 and so on.

Source code for Binary search Tree. Let us consider a **faulty binary tree which is not a binary search tree**.

**public** **class** FaultyBinaryTree {

**private** Node root;

**public** **class** Node {

**private** Node left; **private** Node right; **private** **int** data;

**public** Node( **int** x ) {

data = x;

}

}

**public** Node insert( **int** x ) {

**return** root = insert(root,x);

}

**private** **int** counter = 0;

**public** Node insert(Node node , **int** x ) {

counter++;

**if**( node == **null** ) node = **new** Node(x);

**else** **if**( counter%2 == 0 ) node.left = insert(node.left,x);

**else**

node.right = insert(node.right,x);

**return** node;

}

**public** **void** print() {

printTree(root);

}

**public** **void** printTree(Node node) {

**if**( node == **null** ) **return**;

printTree(node.left);

System.*out*.print(node.data+" ");

printTree(node.right);

}

**int** lastValue = Integer.*MIN\_VALUE*;

**public** **boolean** isValidBST(Node node) {

**if** (node == **null**) **return** **true**;

**if** (!isValidBST(node.left)) **return** **false**;

**if** (node.data<lastValue) **return** **false**;

lastValue=node.data;

**return** isValidBST(node.right);

}

**public** **boolean** isValidBST() {

**return** isValidBST(root);

}

**public** **static** **void** main(String[] args) {

FaultyBinaryTree bt = **new** FaultyBinaryTree();

bt.insert(11);

bt.insert(20);

bt.insert(12);

bt.insert(21);

bt.print();

System.*out*.println("\nIs Binary Search Tree : "+bt.isValidBST());

}

}

**Complete and Correct Binary search tree code is given below.**

**public** **class** BinaryTree {

**private** Node root;

**public** **class** Node {

**private** Node left;

**private** Node right;

**private** **int** data;

**public** Node(**int** x) {

data = x;

left = **null**;

right = **null**;

}

}

**public** Node insert(**int** x) {

**return** root = insert(root,x);

}

//BST should not have duplicate

**public** Node insert(Node node , **int** x) {

**if**( node == **null** )

node = **new** Node(x);

**else** {

**if**( x < node.data)

node.left = insert(node.left,x);

**else** **if**(x > node.data)

node.right = insert(node.right,x);

**else**

node.data = x;

}

**return** node;

}

**public** **boolean** isBST() {

**return** isBST(root, Integer.*MIN\_VALUE*, Integer.*MAX\_VALUE*);

}

**public** **boolean** isBST(Node node , **int** min , **int** max) {

**if** (node==**null**)

**return** **true**;

**if** (node.data < min || node.data > max)

**return** **false**;

**return**

isBST(node.left, min, node.data-1) && isBST(node.right, node.data+1, max);

}

**public** **boolean** isValidBST() { //Another way of writing

**return** isValidBST(root);

}

**int** lastValue = Integer.*MIN\_VALUE*;

**public** **boolean** isValidBST(Node root) {

**if** (root == **null**) **return** **true**;

**if** (!isValidBST(root.left)) **return** **false**;

**if** (root.data<lastValue) **return** **false**;

lastValue=root.data;

**return** isValidBST(root.right);

}

**public** **void** print() {

printTree(root);

}

**public** **void** printTree(Node node) {

**if**( node == **null** )

**return**;

printTree(node.left);

System.*out*.print(node.data+" ");

printTree(node.right);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\* Binary Tree Traversal \*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **void** preOrderVisit() {

preOrderVisit(root);

}

**public** **void** preOrderVisit(Node node) {

**if**( node == **null** ) **return**;

System.*out*.print(node.data+" ");

preOrderVisit(node.left);

preOrderVisit(node.right);

}

**public** **void** postOrderVisit() {

postOrderVisit(root);

}

**public** **void** postOrderVisit(Node node) {

**if**( node == **null** ) **return**;

postOrderVisit(node.left);

postOrderVisit(node.right);

System.*out*.print(node.data+" ");

}

**public** **void** inOrderVisit() {

inOrderVisit(root);

}

**public** **void** inOrderVisit(Node node) {

**if**( node == **null** ) **return**;

inOrderVisit(node.left);

System.*out*.print(node.data+" ");

inOrderVisit(node.right);

}

//Find minimum value

**public** **int** findMinimum(){

**if** ( root == **null** ){

**return** 0;

}

Node currNode = root;

**while**(currNode.left != **null**){

currNode = currNode.left;

}

**return** currNode.data;

}

**public** **int** findMin() { //Another way of writing minimum

**int** val = 0;

**for**( Node p = root ; p != **null**; p = p.left )

val = p.data;

**return** val;

}

//Find Maximum

**public** **int** findMaximum() {

**if** ( root == **null**){

**return** 0;

}

Node currNode = root;

**while**(currNode.right != **null**){

currNode = currNode.right;

}

**return** currNode.data;

}

**public** **int** findMax() { //Another way of writing maximum

**int** val = 0;

**for**( Node p = root ; p != **null**; p = p.right )

val = p.data;

**return** val;

}

**public** **int** minDepth() {

**return** minDepth(root);

}

**public** **int** minDepth(Node node) {

**if**( node == **null** ) **return** 0;

Node current = node;

**while**( current.left != **null**) {

current = current.left;

}

**return** current.data;

}

**public** **int** maxDepth() {

**return** maxDepth(root);

}

**public** **int** maxDepth(Node node) {

**if**( node == **null** ) **return** 0;

**else** {

**int** lDepth = maxDepth(node.left);

**int** rDepth = maxDepth(node.right);

**return** ( Math.*max*(lDepth, rDepth)+1 ) ;

}

}

**public** **int** maxDepth1() {

**return** maxDepth(root);

}

**public** **int** maxDepth1(Node node) {

**if**( node == **null** ) **return** 0;

**return** ( Math.*max*(maxDepth(node.left), maxDepth(node.right))+1 ) ;

}

**public** **int** minDepth1() {

**return** minDepth1(root);

}

**public** **int** minDepth1(Node node){

**if**(node == **null**) **return** 0;

**return** (Math.*min*( minDepth(node.left), minDepth(node.right) ) + 1);

}

**public** **int** minDepth2() {

**return** minDepth2(root);

}

**public** **int** minDepth2(Node node) {

**if** (node == **null**)

**return** 0;

**int** left = minDepth(node.left);

System.*out*.println("left : "+left);

**int** right = minDepth(node.right);

System.*out*.println("right : "+right);

**if** (0 == left && 0 == right) **return** 1;

**if** (0 == left || 0 == right)

**return** Math.*max*(left, right) + 1;

**return** Math.*min*(left, right) + 1;

}

//Balanced Binary search

**public** **boolean** isBalanced() {

**return** isBalanced(root);

}

**public** **boolean** isBalanced(Node node) {

**return** (maxDepth(root) - minDepth(root) <= 1);

}

//Height of Binary search tree

**public** **int** getHeight() {

**return** getHeight(root);

}

**public** **int** getHeight(Node node) {

**if** (node == **null**)

**return** -1;

**else**

**return** Math.*max*(getHeight(node.left),getHeight(node.right))+1;

}

//Returns the number of nodes in the tree

**public** **int** size() {

**return** size(root);

}

**public** **int** size(Node node) {

**if**( node == **null** ) **return** 0;

**else**

**return** ( size(node.left) + 1 + size(node.right) );

}

**public** **void** mirror() {

mirror(root);

}

**public** **void** mirror(Node node) {

**if**( node != **null** ) {

mirror(node.left);

mirror(node.right);

Node temp = node.left;

node.left = node.right;

node.right = temp;

}

}

**public** Node delete(**int** x) {  
 **return root** = delete(**root**, x);  
 }  
  
**public** Node delete(Node node, **int** x) {  
 **if** (x < node.**data**)  
 node.**left** = delete(node.**left**, x);  
 **else if** (x > node.**data**)  
 node.**right** = delete(node.**right**, x);  
 **else** {  
 **if** (node.**left** == **null**) **return** node.**right**;  
 **else if** (node.**right** == **null**) **return** node.**left**;  
 **else** {  
 node.**data** = node.**left**.**data**;  
 **node.left = delete(node.left, node.data);** }  
 }  
 **return** node;  
}  
 **public boolean** search(**int** x) {  
 **return** search(**root**, x);  
 }  
  
 **public boolean** search(Node node, **int** x) {  
  
 **if** (x < node.**data**)  
 **return** search(node.**left**, x);  
 **else if** (x > node.**data**)  
 **return** search(node.**right**, x);  
 **else  
 return false**;  
 }

**public void depthFirstSearch() {**

**Queue<Node> q = new LinkedList<Node>();**

**q.offer(root);**

**while (!q.isEmpty()) {**

**Node node = q.poll();**

**System.out.print("\t" + node.data);**

**if (node.left != null)**

**q.offer(node.left);**

**if (node.right != null)**

**q.offer(node.right);**

**}**

**}**

**public** **static** **void** main(String[] args) {

BinaryTree bt = **new** BinaryTree();

bt.insert(11);

bt.insert(7); bt.insert(13); bt.insert(10);

bt.print();

System.*out*.println("\n");

bt.preOrderVisit();

System.*out*.println("\n");

bt.postOrderVisit();

System.*out*.println("\n");

bt.inOrderVisit();

System.*out*.println(bt.isValidBST());

System.*out*.println("Minimum---->"+bt.findMinimum());

System.*out*.println("Minimum---->"+bt.findMin());

System.*out*.println("Maximum : "+bt.findMaximum());

System.*out*.println("Maximum : "+bt.findMax());

//Follows Builder design pattern

System.*out*.println("Min Depth : "+bt.minDepth());

System.*out*.println("Max Depth : "+bt.maxDepth());

System.*out*.println("Max Depth : "+bt.maxDepth1());

System.*out*.println("Min Depth : "+bt.minDepth1());

System.*out*.println("Min Depth : "+bt.minDepth2());

System.*out*.println("is Balanced : "+bt.isBalanced());

System.*out*.println("Height : "+bt.getHeight());

System.*out*.println("Size : "+bt.size());

bt.mirror();

System.*out*.println("\n");

bt.print();

}

}

Breadth-First Traversal in BST

This can be achieved using breadth first traversal algorithm.  
– push level wise nodes into the queue at every iteration  
– pop the node from queue at every iteration and print its value

Lets look at example:

|  |  |
| --- | --- |
| 1  2  3 | 20      15              25  12      18      22      28 |

BFS puts this tree in queue such there level wise nodes are placed one-another like this

|  |  |
| --- | --- |
| 1  2  3 | --------------------------------  20 | 15 | 25 | 12 | 18 | 22 | 28  -------------------------------- |

And at every loop element from queue is removed in FIFO manner

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public void breadthFirstNonRecursive() {      Queue<BinaryNode> queue = new java.util.LinkedList<BinaryNode>();      queue.offer(root);      while (!queue.isEmpty()) {          BinaryNode node = queue.poll();          System.out.println(node.element);          if (node.left != null)              queue.offer(node.left);          if (node.right != null)              queue.offer(node.right);      }  } |

**Output:**20, 15, 25, 12, 18, 22, 28

<http://tekmarathon.com/2013/05/02/print-level-wise-nodes-in-the-binary-tree-breadth-first-traversal/>

Binary Search Tree

import java.util.LinkedList;

import java.util.Queue;

public class BST {

private Node root;

private static class Node {

private Node left;

private Node right;

private int data;

public Node(int x) {

data = x;

left = null;

right = null;

}

}

public Node add(int x) {

return root = insert(root, x);

}

public Node insert(Node node, int x) {

if (node == null)

node = new Node(x);

else if (x < node.data)

node.left = insert(node.left, x);

else if (x > node.data)

node.right = insert(node.right, x);

else

node.data = x;

return node;

}

public Node delete(int x) {

return root = delete(root, x);

}

//Correct Implementation of delete method

**public Node delete(Node node, int x) {**

**if (node == null) throw new RuntimeException("No data available ");**

**else if (x < node.data)**

**node.left = delete(node.left, x);**

**else if (x > node.data)**

**node.right = delete(node.right, x);**

**else {**

**if (node.left == null) return node.right;**

**else if (node.right == null) return node.left;**

**else {**

**node.data = node.left.data;**

**// get data from the rightmost node in the left subtree**

**// node.data = retrieveData(node.left);**

**node.left = delete(node.left, node.data);**

**}**

**}**

**return node;**

**}**

private int retrieveData(Node node) {

while (node.right != null) node = node.right;

return node.data;

}

**public boolean search(int x) {**

**return search(root, x);**

**}**

**public boolean search(Node node, int x) {**

**if (node == null) return false;**

**else if (x < node.data)**

**return search(node.left, x);**

**else if (x > node.data)**

**return search(node.right, x);**

**else**

**return true;**

**}**

public void depthFirstSearch() {

Queue<Node> q = new LinkedList<Node>();

q.offer(root);

while (!q.isEmpty()) {

Node node = q.poll();

System.out.print("\t" + node.data);

if (node.left != null)

q.offer(node.left);

if (node.right != null)

q.offer(node.right);

}

}

public static void main(String[] args) {

int[] a = new int[]{2, 23, 45, -9, 11, -13, 22, 18, 7, 9};

// int[] a = new int[]{2, 23, 45, 11, 22, 18, 7, 9};

BST bt = new BST();

for (int i : a)

bt.add(i);

System.out.println("\n\*\*\*\*\*\*\*\*\*\*\* DFS \*\*\*\*\*\*\*\*\n");

bt.depthFirstSearch();

}

}

Miscellaneous Algorithm

**Write a thread-safe array-based queue implementation in java**

import java.util.ArrayList;

/\*

\* Write a thread-safe array-based queue implementation in java.

\* If one thread reaches to a limit, it should wait for dequeue thread to create a space and vice-versa.

\*/

public class ThreadSafeArrayQueue<E> {

private static final int size=10;

private ArrayList<E> arr= new ArrayList<E>(10);

public synchronized void enqueue(E item) throws InterruptedException

{

while(arr.size() == size){

wait();

}

if(arr.isEmpty())

{

notify();

}

System.out.println("Item added : "+item);

arr.add(item);

}

public synchronized E dequeue(int item) throws InterruptedException

{

while(arr.isEmpty())

{

wait();

}

if(arr.size()==size)

{

notify();

}

return arr.remove(0);

}

public static void main(String[] args) throws Exception {

ThreadSafeArrayQueue<String> thSafeQ = new ThreadSafeArrayQueue<String>();

for( int i = 0 ; i < 11 ; i++ )

thSafeQ.enqueue("Item-"+i);

}

}

**Swap two numbers without using third variable**

/\*

\* Swap two numbers without using third variable

\*/

public class SwapTwoNumbers {

//Using XOR operation

public static void swapTwoNumbers( int a , int b ) {

System.out.println(a+"---"+b);

**a = a ^ b;**

**b = a ^ b;**

**a = a ^ b;**

System.out.println(a+"---"+b);

}

//Using Mathematical operation

public static void swapTwoNumbers1( int a , int b ) {

System.out.println(a+"---"+b);

a = a+b;

b = a-b;

a = a-b;

System.out.println(a+"---"+b);

}

public static void swapTwoNumbers2( int a , int b ) {

System.out.println(a+"---"+b);

a^=b;

b^=a;

a^=b;

System.out.println(a+"---"+b);

}

public static void swapTwoNumbers3( int a , int b ) {

System.out.println(a+"---"+b);

a=a\*b;

b=a/b;

a=a/b;//It will throw Arithmatic Exception in case of 0.

System.out.println(a+"---"+b);

}

public static void main(String[] args) {

int a = 10;//0

int b = 20;

swapTwoNumbers(a, b);

swapTwoNumbers1(a, b);

swapTwoNumbers2(a, b);

}

}

**SORT R G B in a particular order**

/\*

\* Given a character array as input.

\* Array contains only three types of characters 'R', 'G' and 'B'.

\* Sort the array such that all 'R's comes before 'G's and all 'G's comes before 'B's.

\* Constraint :- No extra space allowed(except O(1) space like variables) and minimise the time complexity.

\*

\* Input: R G B G B G B R R B G R R B B G R

\*

\* Output: R R R R R R G G G G G B B B B B B

\*/

**public** **class** SortRGB {

**public** **static** **void** sortAnArrayInParticularOrder(**char** [] array)

{

**int** indexOfR = 0;

**int** indexOfB = array.length-1;

**for**(**int** i=0;i<=indexOfB;)

{

**if**(array[i]=='R')

{

*swap*(i,indexOfR,array);

indexOfR ++;

i++;

}

**else** **if** (array[i]=='B')

{

*swap*(i,indexOfB,array);

indexOfB --;

}

**else**

{

i++;

}

}

**for**(**char** c:array)

{

System.*out*.print(c+" ");

}

}

**private** **static** **void** swap(**int** i, **int** j, **char**[] array) {

**char** temp = array[i];

array[i]=array[j];

array[j]=temp;

}

**public** **static** **void** main(String[] args) {

String s = "RGBGBGBRRBGRRBBGR";

*sortAnArrayInParticularOrder*(s.toCharArray());

}

}

**Calculate the size of a serialized object**

import java.io.ByteArrayOutputStream;

import java.io.ObjectOutputStream;

import java.util.ArrayList;

import java.util.List;

/\*

\* How to calculate the size of a serialized object

\*/

public class SizeOfAnObject {

public static void main(String[] args) {

try {

List<String> list = new ArrayList<String>();

list.add("ABCD");

ByteArrayOutputStream baos = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(baos);

oos.writeObject(list);

oos.close();

System.out.println("Size of List : " + baos.size());// 65

} catch (Exception e) {

e.printStackTrace();

}

}

}

Array Based Stack in java  
  
**import** java.lang.reflect.Array;  
**import** java.util.Arrays;  
**public class** ArrayBasedStack<E> { *//LIFO*  
 **private int defaultCapacity** = 10;  
 **private int size**;  
 **private** Object[] **elements**;  
  
 **public** ArrayBasedStack() {  
 **elements** = **new** Object[**defaultCapacity**] ;  
 }  
  
 **private void** ensureCapacity() {  
 **elements** = Arrays.*copyOf*( **elements**, **size**\*2 );  
 }  
  
 **public void** push( E e ) {  
 **if**( **size** == **elements**.**length**)  
 ensureCapacity();  
 **elements**[**size**++] = e;  
 }  
  
 **public int** getSize() {  
 **return size**;  
 }  
  
 **public** E pop() {  
 E e = (E) **elements**[ --**size** ];  
 **elements**[**size**] = **null**;  
 **return** e;  
 }  
  
 **public** E getElement( **int** index ) {  
 **return** (E) **elements**[index];  
 }  
  
 **public static void** main(String[] args) {  
 ArrayBasedStack stack = **new** ArrayBasedStack();  
 stack.push(**"1"**); stack.push(**"4"**);  
 System.***out***.println(stack.pop());  
 System.***out***.println(stack.pop());  
 **for** (**int** i = 5; i < 25; i++) {  
 stack.push( String.*valueOf*(i) );  
 }  
 **for** (**int** i = 0; i < stack.getSize(); i++) {  
 System.***out***.println(**"Element --->"**+stack.getElement(i));  
 }  
 }  
}

Array Based Queue

**import** java.util.Arrays;  
**public class** ArrayBasedQueue<E> { *// FIFO* **private int defaultCapacity** = 10;  
 **private int size**;  
 **private** Object[] **elements**;  
  
 **public** ArrayBasedQueue() {  
 **elements** = **new** Object[**defaultCapacity**];  
 }  
  
 **private void** ensureCapacity() {  
 **elements** = Arrays.*copyOf*(**elements**, **size** \* 2);  
 }  
  
 **public void** offer(E e) {  
 **if** (**size** == **elements**.**length**)  
 ensureCapacity();  
 **elements**[**size**++] = e;  
 }  
  
 **public int** getSize() {  
 **return size**;  
 }  
  
 **private int front** = 0;  
 **public** E poll() {  
 **front**++;  
 E e = (E) **elements**[**front** - 1];  
 **size**--;  
 **return** e;  
 }  
  
 **public** E getElement(**int** index) {  
 **return** (E) **elements**[index];  
 }  
  
 **public static void** main(String[] args) {  
 ArrayBasedQueue queue = **new** ArrayBasedQueue();  
 queue.offer(**"1"**);  
 queue.offer(**"4"**);  
 System.***out***.println(queue.poll());  
 System.***out***.println(queue.poll());  
 **for** (**int** i = 5; i < 25; i++) {  
 queue.offer(String.*valueOf*(i) );  
 }  
 **for** (**int** i = 0; i < queue.getSize(); i++)

System.out.println("Element --->" + queue.getElement(i));  
 }  
}

Node based Queue

**public class** NodeBasedQueue {  
  
 **private** Node **front**;  
 **private** Node **back**;  
  
 **private static class** Node {  
 **public** Object **element**;  
 **public** Node **next**;  
  
 **public** Node(Object element) {  
 **this**.**element** = element;  
 **next** = **null**;  
 }  
 }  
  
 **public void** offer(Object x) { *//enqueue()* **if**( **front** == **null** )  
 **back** = **front** = **new** Node(x);  
 **else  
 back** = **back**.**next** = **new** Node(x);  
 }  
  
 **public** Object poll() {  
  
 **if**( **front** == **null** ) **throw new** NullPointerException(**"No element found"**);  
  
 Object returnValue = **front**.**element**;  
 **front** = **front**.**next**;  
 **return** returnValue;  
 }  
  
 **public** Object getFront() {  
 **if**( **front** == **null** ) **throw new** NullPointerException(**"No element found"**);  
  
 **return front**.**element**;  
 }  
  
 **public void** makeQueueEmpty() {  
 **front** = **null**;  
 **back** = **null**;  
 }  
  
 **public void** print() {  
 **for**( Node p = **front** ; p != **null** ; p = p.**next**)  
 System.***out***.println(**"Element --->"**+p.**element**);  
 }

**public static void** main(String[] args) {  
  
 NodeBasedQueue queue = **new** NodeBasedQueue();  
 queue.offer(**"1"**);  
 queue.offer(**"2"**);  
 queue.offer(**"3"**);  
 queue.offer(**"4"**);  
  
 System.***out***.println(queue.poll());  
 System.***out***.println(queue.poll());  
  
 queue.print();  
 }  
}

Node based Stack

**public class** NodeBasedStack {  
 **private** Node **front**;  
 **private** Node **back**;  
  
 **private static class** Node {  
 **public** Object **element**;  
 **public** Node **next**;  
  
 **public** Node(Object element) {  
 **this**.**element** = element;  
 **next** = **null**;  
 }  
 }  
  
 **public void** push(Object x) { *//enqueue()* **if**( **front** == **null** ) **back** = **front** = **new** Node(x);  
 **else  
 back** = **back**.**next** = **new** Node(x);  
 }  
  
 **public** Object pop() {  
 **if**( **front** == **null** ) **throw new** NullPointerException(**"No element found"**);  
  
 Object returnValue = **back**.**element**;  
 **if**( **front**.equals(**back**))  
 **front** = **back** = **null**;  
 **else** {  
 Node current = **front**;  
 **while**( current.**next** != **back** )  
 current = current.**next**;  
 **back** = current;  
 current.**next** = **null**;  
 }  
 **return** returnValue;  
 }  
  
 **public** Object getFront() {  
 **if**( **front** == **null** ) **throw new** NullPointerException(**"No element found"**);  
 **return front**.**element**;  
 }  
  
 **public void** makeQueueEmpty() {  
 **front** = **null**;  
 **back** = **null**;  
 }  
  
 **public void** print() {  
 **for**( Node p = **front** ; p != **null** ; p = p.**next**)  
 System.***out***.println(**"Element --->"**+p.**element**);  
 }  
  
 **public static void** main(String[] args) {  
  
 NodeBasedStack stack = **new** NodeBasedStack();  
 stack.push(**"1"**);  
 stack.push(**"2"**);  
 stack.push(**"3"**);  
 stack.push(**"4"**);  
  
 System.***out***.println(stack.pop());  
 System.***out***.println(stack.pop());  
  
 **for** (**int** i = 5; i < 25; i++) {  
 String val = String.*valueOf*(i);  
 stack.push(val);  
 }  
  
 stack.print();  
 }  
}

**Boyer-Moore Algorithm**

**public class** BoyerMooreAlgorithm {  
  
 **private final int BASE**;  
 **private int**[] **occurance**;  
 **private** String **pattern**;  
  
 **public** BoyerMooreAlgorithm(String pattern) {  
 **this**.**BASE** = 256;  
 **this**.**pattern** = pattern;  
 **occurance** = **new int**[**BASE**];  
 **for** (**int** c = 0; c < **BASE**; c++)  
 **occurance**[c] = -1;  
 **for** (**int** j = 0; j < pattern.length(); j++)  
 **occurance**[pattern.charAt(j)] = j;  
 }  
  
 **public int** search(String text) {  
 **int** n = text.length();  
 **int** m = **pattern**.length();  
 **int** skip;  
  
 **for** (**int** i = 0; i <= n - m; i += skip) {  
  
 skip = 0;  
 **for** (**int** j = m - 1; j >= 0; j--) {  
 **if** (**pattern**.charAt(j) != text.charAt(i + j)) {  
 skip = Math.*max*(1, j - **occurance**[text.charAt(i + j)]);  
 **break**;  
 }  
 }  
 **if** (skip == 0) **return** i;  
  
 }  
 **return** n;  
 }  
  
 **public static void** main(String[] args) {  
 String text = **"Lorem ipsum dolor sit amet"**;  
 String pattern = **"ipsum"**;  
 BoyerMooreAlgorithm bm = **new** BoyerMooreAlgorithm(pattern);  
 **int** firstPosition = bm.search(text);  
 System.***out***.println(**"Position :::"**+firstPosition);  
 }  
}

**Another variation of Boyer-Moore Algorithm**

**import** java.util.Arrays;  
  
**public class** BoyerMooreAlgorithm {  
  
 **public static int** search(String text, String pat) {  
 **int** val = -1;  
 **int**[] dummy = **new int**[256];  
 Arrays.*fill*(dummy, -1);  
 **for** (**int** j = 0; j < pat.length(); j++) {  
 dummy[pat.charAt(j)] = j;  
 }  
  
 **int** m = text.length();  
 **int** n = pat.length();  
  
 **int** skip;  
  
 **for** (**int** i = 0; i <= m - n; i += skip) {  
 skip = 0;  
 **for** (**int** j = n - 1; j >= 0; j--) {  
  
 **if** (pat.charAt(j) != text.charAt(i + j)) {  
 skip = Math.*max*(1, j - dummy[text.charAt(i + j)]);  
 **break**;  
 }  
 }  
 **if** (skip == 0) **return** i;  
 }  
 **return** val;  
 }  
  
 **public static void** main(String[] args) {  
 String text = **"Lorem ipsum dolor sit amet"**;  
 String pattern = **"ipsum"**;  
 BoyerMooreAlgorithm bm = **new** BoyerMooreAlgorithm();  
 **int** firstPosition = bm.*search*(text,pattern);  
 System.***out***.println(**"Position :::"** + firstPosition);  
 }  
}

**Brute Force Algorithm**

The brute force algorithm consists in checking, at all positions in the text between 0 and *n*-*m*, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right. The brute force algorithm requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. During the searching phase the text character comparisons can be done in any order. The time complexity of this searching phase is ***O***(*mn*)

**public class** BruteForceAlgorithm {  
  
 **public static int** search( String txt , String pat ) {  
  
 **int** M = pat.length();  
 **int** N = txt.length();  
  
 **for** (**int** i = 0; i <= N - M ; i++) {  
 **int** j;  
 **for** (j = 0; j < M; j++) {  
 **if**( txt.charAt(i+j) != pat.charAt(j))  
 **break**;  
 }  
 **if**( j == M) **return** i;  
 }  
 **return** -1;  
 }  
  
 **public static void** main(String[] args) {  
  
 **int** result = *search*(**"Hello world"** ,**"o wo"**);  
 System.***out***.println(result);  
 }  
}

**Find the common elements between two arrays**

import java.util.ArrayList;

import java.util.Hashtable;

import java.util.LinkedList;

public class CommonElement {

public static void main(String[] args)

{

int arr1[] ={2,17,3,31,4,22,5,9};

int arr2[] = {15,19,22,3,4,6,12,15,19,22,17};

LinkedList link = new LinkedList();

Hashtable h = new Hashtable();

for(int i=0;i<arr1.length;i++)

{

h.put(i,arr1[i]);

}

for(int j=0;j<arr2.length;j++)

{

if(h.contains(arr2[j])){

link.add(arr2[j]);

}

}

System.out.println(link);

}

}

**or**

import java.util.\*;

public class Main

{

public static void main(String[] args)

{

String[] strings1={"a","b","b","c"},

strings2={"b","c","c","d"};

Set<String> set=new LinkedHashSet<String>(Arrays.asList(strings1));

set.retainAll(Arrays.asList(strings2));

System.out.println(set);

}

}

**or**

There are quite a few ways to find the common elements of two arrays (they do not have to have the same lengths). The worst one is O(MxN) with brutal force, and a pretty beautiful one would be dynamic planning. We can sort the two arrays first and do something similar as the final step of a mergesort. I have written my own code to implement this.

public class Intersect

{

static int [] a = {1, 2, 3, 3, 5, 6, 7, 7, 7, 8};

static int [] b = {2, 2, 2, 2, 4, 5, 8, 10};

public static void main(String args[])

{

int j = 0;

for(int i = 0; i < a.length; i++)

{

while(i + 1 < a.length && a[i + 1] == a[i])

{

i++;

}

while(j + 1 < a.length && b[j + 1] == b[j])

{

j++;

}

if(a[i] == b[j])

{

System.out.print(a[i] + " ");

}

else if(b[j] < a[i])

{

j++;

}

}

}

}

**or**

public class CommonElementsInArray

{

public static void main(String a[])

{

int[] arr1 = {4,7,3,9,2};

int[] arr2 = {3,2,12,9,40,32,4};

for(int i=0;i<arr1.length;i++)

{

for(int j=0;j<arr2.length;j++)

{

if(arr1[i]==arr2[j])

{

System.out.println(arr1[i]);

}

}

}

}

}

**Anagram – Two Strings are anagram or not**

<http://stackoverflow.com/questions/15045640/how-to-check-if-two-words-are-anagrams>

Fastest algorithm would be to map each of the 26 English characters to a unique prime number. Then calculate the product of the string. By the fundamental theorem of arithmetic, 2 strings are anagrams if and only if their products are the same.

**public class** Anagram {  
  
 **public static void** main(String[] args) {  
  
 String s1 = **"post"**;  
 String s2 = **"pots"**;  
  
 **int** val1 = 1;  
 **int** val2 = 1;  
 **for** (**int** i = 0; i < s1.length(); i++) {  
 val1 = val1\*s1.charAt(i);  
 val2 = val2\*s2.charAt(i);  
 }  
  
 System.***out***.println(val1+**"-----"**+val2);  
 }  
}

Write your own iterator for an array

Class name is ArrayIterator  
  
**import** java.util.ArrayList;  
**import** java.util.Iterator;  
  
**public class** ArrayIterator **implements** Iterable {  
  
 **private** Object[] **elements**;  
 **private int size**;  
 **private int counter** = 0;  
  
 **public** ArrayIterator() {  
  
 **elements** = **new** Object[20];  
 }  
  
 **public void** add( Object x ) {  
 **elements**[**size**++] = x;  
 }  
  
 **@Override  
 public Iterator iterator() {  
 *//reset the counter  
 //If you do not reset the counter, you will not be iterate once again* counter = 0;  
 return new MyIterator();  
 }**  
 **private class MyIterator implements Iterator {  
 @Override  
 public boolean hasNext() {  
 return counter < elements.length && elements[counter] != null ;  
 }  
  
 @Override  
 public Object next() {  
 return elements[counter++] ;  
 }  
  
 @Override  
 public void remove() {  
 System.*out*.println("Don't want to delete item");  
 }  
 }**

**public static void** main(String[] args) {  
  
 ArrayIterator arr = **new** ArrayIterator();  
 **for**( **int** i = 0 ; i < 10 ; i++ ) {  
 arr.add( **new** Integer(i));  
 }  
  
 Iterator itr = arr.iterator();  
 **while**( itr.hasNext() ) {  
 System.***out***.print(**"\t"**+itr.next()); *//0 1 2 3 4 5 6 7 8 9* }  
 System.***out***.println(**"\n\n"**);  
 itr = arr.iterator();  
 **while**( itr.hasNext() ) {  
 System.***out***.print(**"\t"** + itr.next()); *//0 1 2 3 4 5 6 7 8 9* }  
  
 *//In case of arraylist also, everytime, you get an iterator,  
 //the index is et to 0 so that it can be iterated.* ArrayList al = **new** ArrayList();  
 **for**( **int** i = 0 ; i < 10 ; i++ ) {  
 al.add( **new** Integer(i));  
 }  
 System.***out***.println(**"\n\n"**);  
 itr = arr.iterator();  
 **while**( itr.hasNext() ) {  
  
 System.***out***.print(**"\t"**+itr.next());  
 }  
  
 System.***out***.println(**"\n\n"**);  
 itr = arr.iterator();  
 **while**( itr.hasNext() ) {  
  
 System.***out***.print(**"\t"**+itr.next());  
 }  
  
 }  
 }

*//public interface Collection<E> extends Iterable<E>  
 //public interface List<E> extends Collection<E>*

**Difference between Iterable and Iterator**

An implementation of Iterable is one that provides an Iterator of itself:

public interface Iterable<T>

{

Iterator<T> iterator();

}

An iterator is a simple way of allowing some to loop through a collection of data without assignment privileges (though with ability to remove).

public interface Iterator<E>

{

boolean hasNext();

E next();

void remove();

}

If a collection is iterable, then it can be iterated using an iterator (and consequently can be used in a for each loop.) The iterator is the actual object that will iterate through the collection. Iterable is an interface which provides Iterator.

**Iterable :** A class that can be iterated over. That is, one that has a notion of "get me the first thing, now the next thing, and so on, until we run out."

**Iterator :** A class that manages iteration over an iterable. That is, it keeps track of where we are in the current iteration, and knows what the next element is and how to get it.

To make an object iterable it needs to emit an Iterator object. To enforce this contract, Iterator interface is to be used. It contains a method named iterator() and it returns Iterator. Hence, any class that implements Iterable will return an Iterator.

public interface Collection<E> extends Iterable<E> {}

HashCode and Equals in an Object

How to write hashcode for an object

**public** **enum** EmpType {

***PERMANENT***,***TEMPORARY***

}

**import** java.util.Arrays;

**public** **class** Employee {

**private** String stringVal;

**private** **byte** byteVal;

**private** **short** shortVal;

**private** **int** intVal;

**private** **float** floatVal;

**private** **double** doubleVal;

**private** **long** longVal;

**private** EmpType enumType; // Permanent or Temporary

**private** **char** ch;

**private** **byte**[] byteArrVal;

**private** String[] strArrVal;

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + Arrays.*hashCode*(byteArrVal);

result = prime \* result + byteVal;

result = prime \* result + ch;

**long** temp;

temp = Double.*doubleToLongBits*(doubleVal);

result = prime \* result + (**int**) (temp ^ (temp >>> 32));

result = prime \* result + ((enumType == **null**) ? 0 : enumType.hashCode());

result = prime \* result + Float.*floatToIntBits*(floatVal);

result = prime \* result + intVal;

result = prime \* result + (**int**) (longVal ^ (longVal >>> 32));

result = prime \* result + shortVal;

result = prime \* result + Arrays.*hashCode*(strArrVal);

result = prime \* result + ((stringVal == **null**) ? 0 : stringVal.hashCode());

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Employee other = (Employee) obj;

**if** (!Arrays.*equals*(byteArrVal, other.byteArrVal))

**return** **false**;

**if** (byteVal != other.byteVal)

**return** **false**;

**if** (ch != other.ch)

**return** **false**;

**if** (Double.*doubleToLongBits*(doubleVal) != Double.*doubleToLongBits*(other.doubleVal))

**return** **false**;

**if** (enumType != other.enumType)

**return** **false**;

**if** (Float.*floatToIntBits*(floatVal) != Float.*floatToIntBits*(other.floatVal))

**return** **false**;

**if** (intVal != other.intVal)

**return** **false**;

**if** (longVal != other.longVal)

**return** **false**;

**if** (shortVal != other.shortVal)

**return** **false**;

**if** (!Arrays.*equals*(strArrVal, other.strArrVal))

**return** **false**;

**if** (stringVal == **null**) {

**if** (other.stringVal != **null**)

**return** **false**;

} **else** **if** (!stringVal.equals(other.stringVal))

**return** **false**;

**return** **true**;

}

}

# **Key points for Hashcode**

**final int prime = 31;**

**int result = 1;**

**Arrays.*hashCode*(byteArrVal); //For byte[]**

**prime \* result + ch; // For char type**

//For double type value

**long temp = Double.*doubleToLongBits*(doubleVal);**

**result = prime \* result + (int) (temp ^ (temp >>> 32));**

//For enum

**result = prime \* result + ((enumType == null) ? 0 : enumType.hashCode());**

**Always remember an enum gives a random value for hashcode.**

//For float type value

**result = prime \* result + Float.*floatToIntBits*(floatVal);**

//For long type value

**result = prime \* result + (int) (longVal ^ (longVal >>> 32));**

//For short type value

**result = prime \* result + shortVal;**

//For String[]

**Arrays.*hashCode*(strArrVal);**

//For String

**result = prime \* result + ((stringVal == null) ? 0 : stringVal.hashCode());**

# **Key Points for equals()**

**public** **boolean** equals(Object obj) {

**Always do the first operation**

1. **if (this == obj) return true;**
2. **if (obj == null) return false;**
3. **if (getClass() != obj.getClass()) return false;**

**Second try to compare each field**

1. Employee other = (Employee) obj;
2. For byte[]

**if (!Arrays.equals(byteArrVal, other.byteArrVal)) return false;**

1. For byte value

**if (byteVal != other.byteVal) return false;**

1. For char type value

**if (ch != other.ch) return false;**

1. For double type value

**if(Double.doubleToLongBits(doubleVal) != Double.doubleToLongBits(other.doubleVal))**

**return false;**

1. For float type value

**if(Float.floatToIntBits(floatVal) != Float.floatToIntBits(other.floatVal))**

**return false;**

1. For enum type

**if (enumType != other.enumType) return false;**

1. For int type value

**if (intVal != other.intVal) return false;**

1. For long type value

**if (longVal != other.longVal) return false;**

1. For short type value

**if (shortVal != other.shortVal) return false;**

1. For String[] type

**if (!Arrays.equals(strArrVal, other.strArrVal)) return false;**

1. For normal String value

**if (stringVal == null) {**

**if (other.stringVal != null)**

**return false;**

**} else if (!stringVal.equals(other.stringVal))**

**return false;**

**Points to Remember**

For Any other Array, use the below

**if (!Arrays.equals(ArrVal, other.ArrVal)) return false;**

For Double type value

**if (Double.doubleToLongBits(doubleVal) != Double.doubleToLongBits(other.doubleVal))**

**return false;**

For float type value

**if(Float.floatToIntBits(floatVal) != Float.floatToIntBits(other.floatVal))**

**return false;**

For char type value

**if (ch != other.ch) return false;**

For enum type value

**if (enumType != other.enumType) return false;**

Reference

The hashcode and equals generated by Intellij Idea is given below for the above class.

**public int** hashCode() {  
 **int** result;  
 **long** temp;  
 result = **stringVal** != **null** ? **stringVal**.hashCode() : 0;  
 result = 31 \* result + (**int**) **byteVal**;  
 result = 31 \* result + (**int**) **shortVal**;  
 result = 31 \* result + **intVal**;  
 result = 31 \* result + (**floatVal** != +0.0f ? Float.*floatToIntBits*(**floatVal**) : 0);  
 temp = Double.*doubleToLongBits*(**doubleVal**);  
 result = 31 \* result + (**int**) (temp ^ (temp >>> 32));  
 result = 31 \* result + (**int**) (**longVal** ^ (**longVal** >>> 32));  
 result = 31 \* result + (**enumType** != **null** ? **enumType**.hashCode() : 0);  
 result = 31 \* result + (**int**) **ch**;  
 result = 31 \* result + (**byteArrVal** != **null** ? Arrays.*hashCode*(**byteArrVal**) : 0);  
 result = 31 \* result + (**strArrVal** != **null** ? Arrays.*hashCode*(**strArrVal**) : 0);  
 **return** result;  
}

**public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
  
 Employee employee = (Employee) o;  
  
 **if** (**byteVal** != employee.**byteVal**) **return false**;  
 **if** (**shortVal** != employee.**shortVal**) **return false**;  
 **if** (**intVal** != employee.**intVal**) **return false**;  
 **if** (Float.*compare*(employee.**floatVal**, **floatVal**) != 0) **return false**;  
 **if** (Double.*compare*(employee.**doubleVal**, **doubleVal**) != 0) **return false**;  
 **if** (**longVal** != employee.**longVal**) **return false**;  
 **if** (**ch** != employee.**ch**) **return false**;  
 **if** (**stringVal** != **null** ? !**stringVal**.equals(employee.**stringVal**) : employee.**stringVal** != **null**) **return false**;  
 **if** (**enumType** != employee.**enumType**) **return false**;  
 **if** (!Arrays.*equals*(**byteArrVal**, employee.**byteArrVal**)) **return false**;  
 *// Probably incorrect - comparing Object[] arrays with Arrays.equals* **return** Arrays.*equals*(**strArrVal**, employee.**strArrVal**);  
  
}

**Example on Negative HashCode**

**class Emp {  
  
 private int sal = Integer.*MAX\_VALUE*;*//It can be MIN\_VALUE* private String name;  
  
 public Emp( String name ) {  
 this.name = name;  
 }**

**}**

**It will give negative hashcode as shown below.**

Emp emp = **new** Emp(**"a"**);  
System.***out***.print(**"Emp HashCode :::"**+emp.hashCode());

Advanced Hashing Concept

<http://java-performance.info/changes-to-string-java-1-7-0_06/>

An original String implementation has 4 non static field: char[] value with string characters, int offset and int count with an index of the first character to use from value and a number of characters to use and int hash with a cached value of a String hash code. As you can see, in a very large number of cases a String will have offset = 0 and count = value.length. The only exception to this rule were the strings created via String.substring calls and all API calls using this method internally (like Pattern.split).

String.substring created a String, which shared an internal char[] value with an original String, which allowed you:

1. To save some memory by sharing character data
2. To run String.substring in a constant time ( O(1) )

At the same time such feature was a source of a possible memory leak: if you extract a tiny substring from an original huge string and discard that original string, you will still have a live reference to the underlying huge char[] value taken from an original String. The only way to avoid it was to call a new String( String ) constructor on such string – it made a copy of a required section of underlying char[], thus unlinking your shorter string from its longer “parent”.

From Java 1.7.0\_06 (as well as in current versions of Java 8 – Nov 13) offset and count fields were removed from a String. This means that you can’t share a part of an underlying char[] valueanymore. Now you can forget about a memory leak described above and never ever use new String(String) constructor anymore. As a drawback, you now have to remember thatString.substring has now a linear complexity instead of a constant one.

**Changes to hashing logic**

There is another change introduced to String class in the same update: a new hashing algorithm. Oracle suggests that a new algorithm gives a better distribution of hash codes, which should improve performance of several hash-based collections: HashMap, Hashtable, HashSet,LinkedHashMap, LinkedHashSet, WeakHashMap and ConcurrentHashMap. Unlike changes from the first part of this article, these changes are experimental and turned off by default.

As you may guess, these changes are only for String keys. If you want to turn them on, you’ll have to set a jdk.map.althashing.threshold system property to a non-negative value (it is equal to-1 by default). This value will be a collection size threshold, after which a new hashing method will be used. A small remark here: hashing method will be changed on rehashing only (when there is no more free space). So, if a collection was rehashed last time at size = 160 andjdk.map.althashing.threshold = 200, then a method will only be changed when your collection will grow to size of 320 (approximately).

String now has a hash32() method, which result is cached in int hash32 field. The biggest difference of this method is that the result of hash32() on the same string may be different on various JVM runs (actually, it will be different in most cases, because it uses a singleSystem.currentTimeMillis() and two System.nanoTime calls for seed initialization). As a result, iteration order on some of your collections will be different each time you run your program.

<http://java-performance.info/hashcode-method-performance-tuning/>

# **hashCode method performance tuning**

In this chapter we will discuss various implications of hashCode method implementation on application performance.

The main purpose of hashCode method is to allow an object to be a key in the hash map or a member of a hash set. In this case an object should also implement equals(Object) method, which is consistent with hashCode implementation:

* If a.equals(b) then a.hashCode() == b.hashCode()
* If hashCode() was called twice on the same object, it should return the same result provided that the object was not changed

#### **hashCode from performance point of view**

From the performance point of view, the main objective for your hashCode method implementation is to minimize the number of objects sharing the same hash code. All JDK hash based collections store their values in an array. Hash code is used to calculate an initial lookup position in this array. After that equals is used to compare given value with values stored in the internal array. So, if all values have distinct hash codes, this will minimize the possibility of hash collisions. On the other hand, if all values will have the same hash code, hash map (or set) will degrade into a list with operations on it having O(n2) complexity.

 JDK is using a method called[open addressing](http://en.wikipedia.org/wiki/Open_addressing), but there is another method called “chaining” – all key-value pairs with the same hash code are stored in a linked list.

**Open addressing**, or **closed hashing**, is a method of [collision resolution in hash tables](https://en.wikipedia.org/wiki/Hash_table#Collision_resolution). With this method a hash collision is resolved by **probing**, or searching through alternate locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.[[1]](https://en.wikipedia.org/wiki/Open_addressing#cite_note-tenenbaum90-1) Well known probe sequences include:

Let’s see the difference in hash code quality. We will compare a normal String with a string wrapper, which overrides hashCode method in order to return the same hash code for all objects.

***private******static******class*** *SlowString*

*{*

***public******final*** *String m\_str;*

***public*** *SlowString(* ***final*** *String str ) {*

***this****.m\_str = str;*

*}*

*@Override*

***public******int*** *hashCode() {*

***return*** *37;*

*}*

*@Override*

***public******boolean*** *equals(Object o) {*

***if*** *(****this*** *== o)* ***return******true****;*

***if*** *(o ==* ***null*** *|| getClass() != o.getClass())* ***return******false****;*

***final*** *SlowString that = ( SlowString ) o;*

***return*** *!(m\_str !=* ***null*** *? !m\_str.equals(that.m\_str) : that.m\_str !=* ***null****);*

*}*

*}*

Here is a testing method. It worth quoting because we will use it again later. It accepts a prepared list of objects (in order not to include these objects creation time in our test) and callsMap.put followed by Map.containsKey on each value in the list.

***private******static******void*** *testMapSpeed(* ***final*** *List lst,* ***final*** *String name )*

*{*

***final*** *Map<Object, Object> map =* ***new*** *HashMap<Object, Object>( lst.size() );*

***int*** *cnt = 0;*

***final******long*** *start = System.currentTimeMillis();*

***for*** *(* ***final*** *Object obj : lst )*

*{*

*map.put( obj, obj );*

***if*** *( map.containsKey( obj ) )*

*++cnt;*

*}*

***final******long*** *time = System.currentTimeMillis() - start;*

*System.out.println( "Time for "  + name + " is " + time / 1000.0 + " sec, cnt = " + cnt );*

*}*

Both String and SlowString objects are created in a loop as "ABCD" + i. It took 0.041 sec to process 100,000 String objects. As for the same number of SlowString objects, it took 82.5 seconds to process them.

As it turned out, String class has exceptional quality hashCode method. Let’s write another test. We will create a list of Strings. First half of them will be equal to "ABCdef\*&" + i, second half –"ABCdef\*&" + i + "ghi" (to ensure that changes in the middle of the string with a constant tail will not decrease hash code quality). We will create 1M, 5M, 10M and 20M strings and see how many of them will share hash codes and how many strings will share the same hash code. This is test output:

*Number of duplicate hashCodes for 1000000 strings = 0*

*Number of duplicate hashCodes for 5000000 strings = 196*

*Number of hashCode duplicates = 2 count = 196*

*Number of duplicate hashCodes for 10000000 strings = 1914*

*Number of hashCode duplicates = 2 count = 1914*

*Number of duplicate hashCodes for 20000000 strings = 17103*

*Number of hashCode duplicates = 2 count = 17103*

So, as you can see, only a very small number of strings is sharing the same hash code and it is very unlikely that one hash code will be shared by more than two strings (unless they are specially crafted, of course). Of course, your data may be different – just run a similar test on your typical keys.

#### **Autogenerated** hashCode **for** long **fields**

It is worth mentioning how hashCode method is generated for long datatype by most of IDEs. Here is a generated hashCode method for a class with 2 long fields:

***public******int*** *hashCode() {*

***int*** *result = (****int****) (val1 ^ (val1 >>> 32));*

*result = 31 \* result + (****int****) (val2 ^ (val2 >>> 32));*

***return*** *result;*

*}*

And here is the similar method generated for a class with 2 int fields:

***public******int*** *hashCode() {*

***int*** *result = val1;*

*result = 31 \* result + val2;*

***return*** *result;*

*}*

As you see, long is treated differently. Similar code is used in java.util.Arrays.hashCode(long a[]). Actually, you will get better hash code distribution if you will extract high and low 32 bits oflong and treat them as int while calculating a hash code. Here is an improved hashCode method for a class with 2 long fields (note that this method runs slower than an original method, but quality of new hash codes will allow hash collections to run faster even at the expense ofhashCode slowdown).

***public******int*** *hashCode() {*

***int*** *result = (****int****) val1;*

*result = 31 \* result + (****int****) (val1 >>> 32);*

*result = 31 \* result + (****int****) val2;*

***return*** *31 \* result + (****int****) (val2 >>> 32);*

*}*

Here are results of testMapSpeed method for processing 10M objects of all three kinds. They were initialized with the same values (all longs actually fit into int values).

| Two longs with original hashCode | Two longs with modified hashCode | Two ints |
| --- | --- | --- |
| 2.596 sec | 1.435 sec | 0.737 sec |

As you can see, hashCode update makes a difference. Not so big, but worth noticing for performance critical code.

#### **Use case: How to benefit from** String.hashCode **quality**

Let’s assume we have a map from string identifiers to some values. Map keys (string identifiers) are not stored anywhere else in memory (at most only some of them may be stored somewhere else at a time). We have already collected all map entries, for example, on the first pass of some two phase algorithm. On the second phase we will need to query map values by keys. We will query our map only using existing map keys.

How can we improve the map? As you have seen before, String.hashCode returns mostly distinct values. We can scan all keys, calculate hash codes of all keys and find not unique hash codes:

*Map<Integer, Integer> cnt =* ***new*** *HashMap<Integer, Integer>( max );*

***for*** *(* ***final*** *String s : dict.keySet() )*

*{*

***final******int*** *hash = s.hashCode();*

***final*** *Integer count = cnt.get( hash );*

***if*** *( count !=* ***null*** *)*

*cnt.put( hash, count + 1 );*

***else***

*cnt.put( hash, 1 );*

*}*

*//keep only not unique hash codes*

***final*** *Map<Integer, Integer> mult =* ***new*** *HashMap<Integer, Integer>( 100 );*

***for*** *(* ***final*** *Map.Entry<Integer, Integer> entry : cnt.entrySet() )*

*{*

***if*** *( entry.getValue() > 1 )*

*mult.put( entry.getKey(), entry.getValue() );*

*}*

Now we can create 2 maps out of the old one. Let’s assume for simplicity that old map values were just Objects. In this case, we will end up with Map<Integer, Object> and Map<String, Object>(for production code Trove TIntObjectHashMap is recommended instead of Map<Integer, Object>). First map will contain mapping from unique hash codes to values, second map – mapping from strings with not unique hash codes to values.

***final*** *Map<Integer, Object> unique =* ***new*** *HashMap<Integer, Object>( 1000 );*

***final*** *Map<String, Object> not\_unique =* ***new*** *HashMap<String, Object>( 1000 );*

*//dict - original map*

***for*** *(* ***final*** *Map.Entry<String, Object> entry : dict.entrySet() )*

*{*

***final******int*** *hashCode = entry.getKey().hashCode();*

***if*** *( mult.containsKey( hashCode ) )*

*not\_unique.put( entry.getKey(), entry.getValue() );*

***else***

*unique.put( hashCode, entry.getValue() );*

*}*

Now, in order to get a value, we need to query unique map first and not\_unique map if first query has not returned a valid result:

***public*** *Object get(* ***final*** *String key )*

*{*

***final******int*** *hashCode = key.hashCode();*

*Object value = m\_unique.get( hashCode );*

***if*** *( value == null )*

*value = m\_not\_unique.get( key );*

***return*** *value;*

*}*

**Hashing function in Map in Java**

//**As of JDK 6**  
static int hash(int h) {

// This function ensures that hashCodes that differ only by  
// constant multiples at each bit position have a bounded  
// number of collisions (approximately 8 at default load factor).

h ^= (h >>> 20) ^ (h >>> 12);  
 return h ^ (h >>> 7) ^ (h >>> 4);  
}

//**As of JDK 7**  
final int hash(Object k) {

int h = hashSeed;  
if (0 != h && k instanceof String) {  
 return sun.misc.Hashing.stringHash32((String) k);  
}  
  
h ^= k.hashCode();  
// This function ensures that hashCodes that differ only by  
// constant multiples at each bit position have a bounded  
// number of collisions (approximately 8 at default load factor).  
  
h ^= (h >>> 20) ^ (h >>> 12);  
 return h ^ (h >>> 7) ^ (h >>> 4);  
}

<https://vaskoz.wordpress.com/2013/04/06/java-7-hashing-drastically-better-than-java-6/>

**Java 7 hashing drastically better than Java 6**

### How does Java 6 hashing work?

java.lang.String#hashCode() is calculated by iterating over each character in the string, and executing this function:

**h = 31\*h + val[i]**

Now let’s examine how the hash code is used by a popular data structure: HashMap. Looking at java.util.HashMap#put(K,V) runs HashMap#hash(int) on the key’s hashCode. That function is a supplemental hash that provides some protection against bad hash function.

**h ^= (h >>> 20) ^ (h >>> 12);**  
**return h ^ (h >>> 7) ^ (h >>> 4);**

This provides no randomization or protection from [**collision attacks**](http://en.wikipedia.org/wiki/Collision_attack). In Java 6, every JVM running everywhere follows these 2 mathematical functions. Some might think, Java hash code should always be the same whenever you call hashCode on the same string, but that’s only true within a single invocation of the JVM. There is no requirement that if 2 different JVM instances running should resolve to the same hashCode for the same string.

### Java 7 finally fixed hashing.

java.lang.String has add a static initializer for the single purpose of creating a**HASHING\_SEED**. This is brand new for Java 7. The “seed materials” used to randomize hash codes so that different JVMs don’t have predictable hash codes are: System.currentTimeMillis and System.nanoTime and others. But wait, if you look at java.lang.String#hashCode() in Java 7 source ***it looks identical to Java 6’s implementation!***So let’s checkout java.util.HashMap in Java 7, and we find that the hash() function there has changed. It takes an Object instead of an int, and if the Object is an instanceof String, then through **sun.misc.Hashing.stringHash32()**it actually called a new method insidejava.lang.String#hash32(). The hash32() function passes the character data plus the HASHING\_SEED computed above to **sun.misc.Hashing.murmur3\_32(HASHING\_SEED, value, 0, value.length)**

The [**Murmur3 hashing function**](http://en.wikipedia.org/wiki/MurmurHash) with the hashing seed data which is based on the time that java.lang.String’s static initializer is executed provides excellent protection against collision attacks.

public class Hash {

public static void main(String[] args) {

String str = "Java Programming Language";

System.out.println(str + " hashCode(): "

+ str.hashCode()

+ " new Java 7 hashCode used by collections: "

+ sun.misc.Hashing.stringHash32(str));

}

}

The above provide two different hashcode, str.hashCode() always gives constant value where as sun.misc.Hashing.stringHash32(str) gives random value.

# **MurmurHash**

<https://en.wikipedia.org/wiki/MurmurHash>

**MurmurHash** is a non-[cryptographic](https://en.wikipedia.org/wiki/Cryptographic_hash_function) [hash function](https://en.wikipedia.org/wiki/Hash_function) suitable for general hash-based lookup. The name comes from two basic operations, multiply (MU) and rotate (R), though the algorithm actually uses shift and xor instead of rotate.

# **Collision attack**

<https://en.wikipedia.org/wiki/Collision_attack>

In cryptography, a **collision attack** on a [cryptographic hash](https://en.wikipedia.org/wiki/Cryptographic_hash) tries to find two inputs producing the same hash value, i.e. a [hash collision](https://en.wikipedia.org/wiki/Hash_collision).

**Collision attack**

Find two different messages *m1* and *m2* such that *hash(m1)* = *hash(m2)*.

## Classical collision attack

Mathematically stated, a collision attack finds two different messages *m1* and *m2*, such that *hash(m1)* = *hash(m2)*. In a classical collision attack, the attacker has no control over the content of either message, but they are arbitrarily chosen by the algorithm.

## Attack scenarios[[edit](https://en.wikipedia.org/w/index.php?title=Collision_attack&action=edit&section=3)]

Many applications of crytographic hash functions do not rely on [collision resistance](https://en.wikipedia.org/wiki/Collision_resistance), thus collision attacks do not affect their security. For example, [HMACs](https://en.wikipedia.org/wiki/HMAC) are not vulnerable.[[9]](https://en.wikipedia.org/wiki/Collision_attack#cite_note-collision-qna-9) For the attack to be useful, the attacker must be in control of the input to the hash function.

### Digital signatures

Because [digital signature](https://en.wikipedia.org/wiki/Digital_signature) algorithms cannot sign a large amount of data efficiently, most implementations use a hash function to reduce ("compress") the amount of data that needs to be signed down to a constant size. Digital signature schemes are often vulnerable to hash collisions, unless using techniques like randomized hashing.[[10]](https://en.wikipedia.org/wiki/Collision_attack#cite_note-10)

Note that all [public key certificates](https://en.wikipedia.org/wiki/Public_key_certificate), like [SSL](https://en.wikipedia.org/wiki/Transport_Layer_Security) certificates, also rely on the security of digital signatures and are compromised by hash collisions.

The usual attack scenario goes like this:

1. Mallory creates two different documents A and B, that have an identical hash value (collision).
2. Mallory then **sends document A to Alice**, who agrees to what the document says, signs its hash and sends it back to Mallory.
3. Mallory copies the signature sent by Alice from document A to document B.
4. Then she **sends document B to Bob**, claiming that Alice signed the other document (document B). Because the digital signature matches the document hash, Bob's software is unable to detect the modification.

Internals of Java Data Structures

**About ArrayList**

**How to get the capacity of ArrayList**

//How to get the capcity of ArrayList using reflection

**static** **int** getCapacity(ArrayList<?> list) **throws** Exception {

Field dataField = ArrayList.**class**.getDeclaredField("elementData");

dataField.setAccessible(**true**);

**return** ((Object[]) dataField.get(list)).length;

}

**How will you get deep copy and shallow of ArrayList**

Shallow copy can be obtained using list.clone() method.

Deep copy can be achieved using serializing and deserializing

ByteArrayOutputStream bos = **new** ByteArrayOutputStream();

ObjectOutputStream o = **new** ObjectOutputStream(bos);

o.writeObject(list);

**byte**[] yourBytes = bos.toByteArray();

InputStream in = **new** ByteArrayInputStream(yourBytes);

ObjectInputStream oin = **new** ObjectInputStream(in);

ArrayList<Employee> deepCopyList = (ArrayList<Employee>) oin.readObject();

**How arrayList grows internally**

**As of JDK 6 or prior to JDK 7**

**int oldCapacity = elementData.length;**

**int newCapacity = (oldCapacity \* 3)/2 + 1;**

**elementData = Arrays.copyOf(elementData, newCapacity);**

public void ensureCapacity(int minCapacity) {

modCount++;

int oldCapacity = elementData.length;

if (minCapacity > oldCapacity) {

Object oldData[] = elementData;

int newCapacity = (oldCapacity \* 3)/2 + 1;

if (newCapacity < minCapacity)

newCapacity = minCapacity;

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

}

**As per JDK 7**

**int oldCapacity = elementData.length;**

**int newCapacity = oldCapacity + (oldCapacity >> 1);**

**elementData = Arrays.copyOf(elementData, newCapacity);**

private void grow(int minCapacity) {

// overflow-conscious code

int oldCapacity = elementData.length;

int newCapacity = oldCapacity + (oldCapacity >> 1);

if (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

newCapacity = hugeCapacity(minCapacity);

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

private static int hugeCapacity(int minCapacity) {

if (minCapacity < 0) // overflow

throw new OutOfMemoryError();

return (minCapacity > MAX\_ARRAY\_SIZE) ? Integer.MAX\_VALUE : MAX\_ARRAY\_SIZE;

}

**How hashcode is calculated in case of ArrayList ?**

The code is given below. It is the part of AbtractList.

**public** **int** hashCode() {

**int** hashCode = 1;

**for** (E e : **this**)

hashCode = 31\*hashCode + (e==**null** ? 0 : e.hashCode());

**return** hashCode;

}

Basically it calculates the hashcode of each object present inside the List.

**How equals method works for ArrayList**

The code is given below. It is the part of AbtractList.

**public** **boolean** equals(Object o) {

**if** (o == **this**)

**return** **true**;

**if** (!(o **instanceof** List))

**return** **false**;

ListIterator<E> e1 = listIterator();

ListIterator e2 = ((List) o).listIterator();

**while** (e1.hasNext() && e2.hasNext()) {

E o1 = e1.next();

Object o2 = e2.next();

**if** (!(o1==**null** ? o2==**null** : o1.equals(o2)))

**return** **false**;

}

**return** !(e1.hasNext() || e2.hasNext());

}

**What is difference between System.arrayCopy and Arrays.copyOf ?**

The difference is that Arrays.copyOf does not only copy elements, it also creates a new array. System.arrayCopy copies into an existing array. Arrays.copyOf also internally uses System.arrayCopy.

**How Serialization and Deserialization happens inside ArrayList**

The code is given below.

**private** **void** writeObject(java.io.ObjectOutputStream s)

**throws** java.io.IOException{

// Write out element count, and any hidden stuff

**int** expectedModCount = modCount;

s.defaultWriteObject();

// Write out size as capacity for behavioural compatibility with clone()

s.writeInt(size);

// Write out all elements in the proper order.

**for** (**int** i=0; i<size; i++) {

s.writeObject(elementData[i]);

}

**if** (modCount != expectedModCount) {

**throw** **new** ConcurrentModificationException();

}

}

/\*\*

\* Reconstitute the <tt>ArrayList</tt> instance from a stream (that is,

\* deserialize it).

\*/

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException {

elementData = ***EMPTY\_ELEMENTDATA***;

// Read in size, and any hidden stuff

s.defaultReadObject();

// Read in capacity

s.readInt(); // ignored

**if** (size > 0) {

// be like clone(), allocate array based upon size not capacity

ensureCapacityInternal(size);

Object[] a = elementData;

// Read in all elements in the proper order.

**for** (**int** i=0; i<size; i++) {

a[i] = s.readObject();

}

}

}

Example of Encapsulation. ArrayList maintains the following buffer statement.

/\*\*

\* The array buffer into which the elements of the ArrayList are stored.

\* The capacity of the ArrayList is the length of this array buffer. Any

\* empty ArrayList with elementData == EMPTY\_ELEMENTDATA will be expanded to

\* DEFAULT\_CAPACITY when the first element is added.

\*/

**private** **transient** Object[] elementData;

There is not set() or get() method for this to ensure security.

**How contains() method works in ArrayList ? In case of null value, how ArrayList’s contains method behaves ?**

The code snippet is given below.

**public** **boolean** contains(Object o) {

**return** indexOf(o) >= 0;

}

**public** **int** indexOf(Object o) {

**if** (o == **null**) {

**for** (**int** i = 0; i < size; i++)

**if** (elementData[i]==**null**)

**return** i;

} **else** {

**for** (**int** i = 0; i < size; i++)

**if** (o.equals(elementData[i]))

**return** i;

}

**return** -1;

}

As you see from the above code, if the object is null or the ArrayList contains many null object, the first null object is obtained, besides it checks the object’s equality. It means we have to override the equals method for the objects to store in the ArrayList.

If do not override equals method, list.contains() return false as shown below.

ArrayList<Employee> list = **new** ArrayList<Employee>();

Employee emp1 = **new** Employee("A");

emp1.setAge(11);

list.add(emp1);

Employee emp2 = **new** Employee("B");

emp2.setAge(12);

list.add(emp2);

Employee emp3 = **new** Employee("A");

emp3.setAge(11);

System.***out***.println(list.contains(emp3));// It returns false

To make the above statement true, we have to override the equals() method. This is the same thing for LinkedList.

**\*\*** **In case LinkedList, the add() methods adds the object in the last of the LinkedList as shown from the code. Appends the specified element to the end of this list.**

**public** **boolean** add(E e) {

linkLast(e);

**return** **true**;

}

**How peek() method works in LinkedList ?**

The code is given below. Retrieves, but does not remove, the head (first element) of this list.

**public** E peek() {

**final** Node<E> f = first;

**return** (f == **null**) ? **null** : f.item;

}

**How element() method works in LinkedList ?**

The code is given below. Retrieves, but does not remove, the head (first element) of this list.

**public** E element() {

**return** getFirst();

}

**How remove() works in LinkedList ?**

The code is given below. Retrieves and removes the head (first element) of this list.

**public** E remove() {

**return** removeFirst();

}

**How offer() method works in LinkedList ?**

The code is given below. Adds the specified element as the tail (last element) of this list.

**public** **boolean** offer(E e) {

**return** add(e);

}

**How clone() method works in LinkedList ?**

The code is given below. Returns a shallow copy of LinkedList.

**public** Object clone() {

LinkedList<E> clone = superClone();

// Put clone into "virgin" state

clone.first = clone.last = **null**;

clone.size = 0;

clone.modCount = 0;

// Initialize clone with our elements

**for** (Node<E> x = first; x != **null**; x = x.next)

clone.add(x.item);

**return** clone;

}

@SuppressWarnings("unchecked")

**private** LinkedList<E> superClone() {

**try** {

**return** (LinkedList<E>) **super**.clone();

} **catch** (CloneNotSupportedException e) {

**throw** **new** InternalError();

}

}

**How Serialization and Deserialization happens in LinkedList ?**

The code is given below.

**private** **void** writeObject(java.io.ObjectOutputStream s)

**throws** java.io.IOException {

// Write out any hidden serialization magic

s.defaultWriteObject();

// Write out size

s.writeInt(size);

// Write out all elements in the proper order.

**for** (Node<E> x = first; x != **null**; x = x.next)

s.writeObject(x.item);

}

@SuppressWarnings("unchecked")

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException {

// Read in any hidden serialization magic

s.defaultReadObject();

// Read in size

**int** size = s.readInt();

// Read in all elements in the proper order.

**for** (**int** i = 0; i < size; i++)

linkLast((E)s.readObject());

}

**\*\* The hashhcode() and equals() method of LinkedList are same as that of AbstractList which are used in ArrayList.**

**Internals of HashSet**

HashSet internally uses HashMap as the code is given below.

**private** **transient** HashMap<E,Object> map;

**private** **static** **final** Object ***PRESENT*** = **new** Object();

The internal HashMap used “PRESENT” as seen above as a dummy object.

In case of LinkedHashSet, the constructor is given below.

**HashSet(int initialCapacity, float loadFactor, boolean dummy) {**

**map = new LinkedHashMap<>(initialCapacity, loadFactor);**

**}**

**The question may be how LinkedHashSet works internally. The answer is LinkedHashSet internally uses LinkedHashMap.**

**The following methods are given about how HashSet internally works.**

**To get size :: map.size();**

**To check hashSet is empty : map.isEmpty();**

**To check whether hashset contains an object : map.containsKey(o);**

**To add object to hashset : map.put(e, *PRESENT*)==null;**

**How Serialization and Deserialization works in HashSet ?**

**The code is given below.**

**private** **void** writeObject(java.io.ObjectOutputStream s)

**throws** java.io.IOException {

// Write out any hidden serialization magic

s.defaultWriteObject();

// Write out HashMap capacity and load factor

s.writeInt(map.capacity());

s.writeFloat(map.loadFactor());

// Write out size

s.writeInt(map.size());

// Write out all elements in the proper order.

**for** (E e : map.keySet())

s.writeObject(e);

}

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException {

// Read in any hidden serialization magic

s.defaultReadObject();

// Read in HashMap capacity and load factor and create backing HashMap

**int** capacity = s.readInt();

**float** loadFactor = s.readFloat();

map = (((HashSet)**this**) **instanceof** LinkedHashSet ?

**new** LinkedHashMap<E,Object>(capacity, loadFactor) :

**new** HashMap<E,Object>(capacity, loadFactor));

// Read in size

**int** size = s.readInt();

// Read in all elements in the proper order.

**for** (**int** i=0; i<size; i++) {

E e = (E) s.readObject();

map.put(e, ***PRESENT***);

}

}

**How clone() method works in HashSet ?**

It returns the shallow copy of the HashSet.

**public** Object clone() {

**try** {

HashSet<E> newSet = (HashSet<E>) **super**.clone();

newSet.map = (HashMap<E, Object>) map.clone();

**return** newSet;

} **catch** (CloneNotSupportedException e) {

**throw** **new** InternalError();

}

}

**How equals() and hashcode() method works in HashSet ?**

The equals method is given below. These methods are part of AbstractSet

**public** **boolean** equals(Object o) {

**if** (o == **this**)

**return** **true**;

**if** (!(o **instanceof** Set))

**return** **false**;

Collection c = (Collection) o;

**if** (c.size() != size())

**return** **false**;

**try** {

**return** containsAll(c);

} **catch** (ClassCastException unused) {

**return** **false**;

} **catch** (NullPointerException unused) {

**return** **false**;

}

}

The hashcode method is given below.

**public** **int** hashCode() {

**int** h = 0;

Iterator<E> i = iterator();

**while** (i.hasNext()) {

E obj = i.next();

**if** (obj != **null**)

h += obj.hashCode();

}

**return** h;

}

How TreeSet works internally

The code is given below.

**private** **transient** NavigableMap<E,Object> m;

**private** **static** **final** Object ***PRESENT*** = **new** Object();

**public** TreeSet() {

**this**(**new** TreeMap<E,Object>());

}

**public** TreeSet(Comparator<? **super** E> comparator) {

**this**(**new** TreeMap<>(comparator));

}

From the above code it is obvious that TreeSet internally uses TreeMap.

\*\* The equals(), clone(), hashcode() methods work like HashSet.

**How LinkedHashSet internally works ?**

LinkedHashSet internally uses HashSet, methods and constructors are inherited.

\*\* TreeSet internally uses TreeMap which has been developed on the base line of Red and Black Tree algorithm.

**How HashMap put() method works internally ?**

Always remember that, the initial capacity and threashold in case of HashMap is 16, 2 << 4.

The load factor is 0.75f.

The constructor for HashMap is given below.

public HashMap() {

this(DEFAULT\_INITIAL\_CAPACITY, DEFAULT\_LOAD\_FACTOR);

}

The source code for put method given below.

**public** V put(K key, V value) {

**if** (table == ***EMPTY\_TABLE***) {

**inflateTable(threshold);**

}

**if** (key == **null**)

**return** putForNullKey(value);

**int hash = hash(key);**

**int i = *indexFor*(hash, table.length);**

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {

Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

**return** **null**;

}

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

**if** ((size >= threshold) && (**null** != table[bucketIndex])) {

**resize(2 \* table.length);**

hash = (**null** != key) ? hash(key) : 0;

bucketIndex = *indexFor*(hash, table.length);

}

createEntry(hash, key, value, bucketIndex);

}

**void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {

Entry<K,V> e = table[bucketIndex];

table[bucketIndex] = **new** Entry<>(hash, key, value, e);

size++;

}

Steps of Put Method

1. Get the hashcode of the Key
2. Calculate the hash value of the hashcode
3. Find the index of the bucket using the following function

**static** **int** indexFor(**int** h, **int** length) {

**return** h & (length-1);

}

1. Iterate the entire bucket and check whether key is already present or not, if present, simply modify the value or update the value.
2. If the key is not present, check the bucket has been filled up or not.
3. If the bucket has been filled up, increase the size of the Entry object array as double, the code is given below.

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

**if** ((size >= threshold) && (**null** != table[bucketIndex])) {

**resize(2 \* table.length);**

hash = (**null** != key) ? hash(key) : 0;

bucketIndex = *indexFor*(hash, table.length);

}

createEntry(hash, key, value, bucketIndex);

}

**void resize(int newCapacity)** {

**Entry[] oldTable = table;**

**int oldCapacity = oldTable.length;**

**if (oldCapacity == *MAXIMUM\_CAPACITY*) {**

**threshold = Integer.*MAX\_VALUE*;**

**return;**

**}**

**Entry[] newTable = new Entry[newCapacity];**

**transfer(newTable, initHashSeedAsNeeded(newCapacity));**

table = newTable;

**threshold = (int)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

}

Let see how it works

* Suppose initial capacity is 16.
* Initial capacity and threshold are same ie 16. Refer to the constructor of hashmap , it

**threshold = initialCapacity;**

**this.loadFactor = loadFactor; // ie 0.75**

* If the size of threshold is greater than or equal to 16 and bucket index has a value, increase the size to double. As it is obvious from the below code.

**if ((size >= threshold) && (null != table[bucketIndex])) {**

**resize(2 \* table.length);**

**In the above case, resize( 2 \* 16 ), so size = 16**

* Now calculate the old capacity which is nothing but the length of the bucket ie

**int oldCapacity = oldTable.length;**

* Check if the old capacity is greater than Maximum Size ie

**static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 30; //**1073741824**

* A new array of size 32 will be created.
* Now new threshold value will be calculated as

**threshold = (int)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

**threshold = (int)Math.*min*(32 \* 0.75, 1073741824 + 1);**

**threshold = (int)Math.*min*(32 \* 0.75, 1073741824 + 1);**

**threshold = (int) Math.min(24, 1073741824 + 1);**

**threshold = 24; //Finally**

How hashmap reacts when you pass the your capacity in hashmap ? How capacity is calculated ?

Let us take an example below.

**HashMap hm = new HashMap(7);**

Now in this case, the initial capacity will be 7, load factor will be 0.75. The code is given below.

**public** HashMap(**int** initialCapacity) {

**this**(initialCapacity, ***DEFAULT\_LOAD\_FACTOR***);

}

**public** HashMap(**int** initialCapacity, **float** loadFactor) {

**if** (initialCapacity < 0)

**throw** **new** IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

**if** (initialCapacity > ***MAXIMUM\_CAPACITY***)

initialCapacity = ***MAXIMUM\_CAPACITY***;

**if** (loadFactor <= 0 || Float.*isNaN*(loadFactor))

**throw** **new** IllegalArgumentException("Illegal load factor: " +

loadFactor);

**this**.loadFactor = loadFactor;

threshold = initialCapacity;

init();

}

Now in this case the growth of HashMap is bit different. Let us see below. Now again see the put method.

**public** V put(K key, V value) {

**if (table == *EMPTY\_TABLE*) {**

**inflateTable(threshold);**

**}**

**if** (key == **null**)

**return** putForNullKey(value);

**int** hash = hash(key);

**int** i = *indexFor*(hash, table.length);

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {

Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

**return** **null**;

}

Let us see the inflateTable() method.

**private** **void** inflateTable(**int** toSize) {

// Find a power of 2 >= toSize

**int capacity = *roundUpToPowerOf2*(toSize);**

**threshold = (int) Math.*min*(capacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

**table = new Entry[capacity];**

initHashSeedAsNeeded(capacity);

}

The above method is used to create initial array of Entry object. Let us see how threshold is calculated.

**int capacity = *roundUpToPowerOf2*(toSize); The method is given below.**

**private** **static** **int** roundUpToPowerOf2(**int** number) {

// assert number >= 0 : "number must be non-negative";

**return** number >= ***MAXIMUM\_CAPACITY***

? ***MAXIMUM\_CAPACITY***

: (number > 1) ? Integer.*highestOneBit*((number - 1) << 1) : 1;

}

What is Integer.*highestOneBit*((number - 1) << 1) : 1; The code is given below.

**public static int** highestOneBit(**int** i) {  
 *// HD, Figure 3-1* i |= (i >> 1);  
 i |= (i >> 2);  
 i |= (i >> 4);  
 i |= (i >> 8);  
 i |= (i >> 16);  
 **return** i - (i >>> 1);  
}

We need to understand about how the above method ***roundUpToPowerOf2*(toSize)** works. Let us consider a small java program and see how numbers are getting generated.

**static final int *MAXIMUM\_CAPACITY*** = 1 << 30;  
**public static int** roundUpToPowerOf2(**int** number) {  
 *// assert number >= 0 : "number must be non-negative";* **return** number >= ***MAXIMUM\_CAPACITY*** ? ***MAXIMUM\_CAPACITY*** : (number > 1) ? Integer.*highestOneBit*((number - 1) << 1) : 1;  
}

System.***out***.println(*roundUpToPowerOf2*(0)); *//1*System.***out***.println(*roundUpToPowerOf2*(1)); *//1*System.***out***.println(*roundUpToPowerOf2*(2)); *//2*System.***out***.println(*roundUpToPowerOf2*(3)); *//4*System.***out***.println(*roundUpToPowerOf2*(4)); *//4*System.***out***.println(*roundUpToPowerOf2*(5)); *//8*System.***out***.println(*roundUpToPowerOf2*(7)); *//8*System.***out***.println(*roundUpToPowerOf2*(9)); *//16*System.***out***.println(*roundUpToPowerOf2*(11)); *//16*System.***out***.println(*roundUpToPowerOf2*(15)); *//16*System.***out***.println(*roundUpToPowerOf2*(16)); *//16*System.***out***.println(*roundUpToPowerOf2*(19)); *//32*System.***out***.println(*roundUpToPowerOf2*(27)); *//32*System.***out***.println(*roundUpToPowerOf2*(32)); *//32*System.***out***.println(*roundUpToPowerOf2*(33)); *//64*System.***out***.println(*roundUpToPowerOf2*(48)); *//64*System.***out***.println(*roundUpToPowerOf2*(65)); *//128*

The mathematical trick is given below.

Choose any number and see how it falls in the below segment.

**… 512 256 128 64 32 16 8 4 2 1**

Let us number 65, it falls in between 64 and 128, so capacity will be 128.

If number is 16, capacity will 16

No threashold calculation

**threshold = (int) Math.*min*(capacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

Let us consider, initial capacity is 5, it falls in between 4 and 8, so capacity will be 8.

initialCapacity = 5;

capacity = 8;

Threshold = (int) Math.min(8 \* 0.75 , ***MAXIMUM\_CAPACITY* + 1)**

Threshold = (int) Math.min(6 , ***MAXIMUM\_CAPACITY* + 1)**

Threshold = 6;

It means after 6 entries, the Entry object will grow and all the values will be copied.

**\*\* When the Entry array object is resized, again the hash value is calculated.**

**In case of LinkedHashMap, the following init() is overridden, but method is blank in case of HashMap. The purpose is to create a LinkedList in case of LinkedHashMap. LinkedHashMap extends HashMap.**

**private** **transient** Entry<K,V> header;

**void** init() {

header = **new** Entry<>(-1, **null**, **null**, **null**);

header.before = header.after = header;

}

**Besides the following methods have been overridden in LinkedHashMap.**

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

**super**.addEntry(hash, key, value, bucketIndex);

// Remove eldest entry if instructed

Entry<K,V> eldest = header.after;

**if** (removeEldestEntry(eldest)) {

removeEntryForKey(eldest.key);

}

}

**void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {

HashMap.Entry<K,V> old = table[bucketIndex];

Entry<K,V> e = **new** Entry<>(hash, key, value, old);

table[bucketIndex] = e;

e.addBefore(header);

size++;

}

**void** transfer(HashMap.Entry[] newTable, **boolean** rehash) {

**int** newCapacity = newTable.length;

**for** (Entry<K,V> e = header.after; e != header; e = e.after) {

**if** (rehash)

e.hash = (e.key == **null**) ? 0 : hash(e.key);

**int** index = *indexFor*(e.hash, newCapacity);

e.next = newTable[index];

newTable[index] = e;

}

}

TreeMap does not extend HashMap, it extends AbstractHashMap only.